Press Release: Duke StatSci First-Year Master Students Develop Enhanced Uplift Modeling Techniques to Improve Marketing Effectiveness

April 26, 2024

Durham, NC – A team of Duke University graduate student has developed an advanced approach to uplift modeling that significantly improves the effectiveness of promotional campaigns. Their study, "A Novel Method for Uplift Modeling with Weights Optimized by Maximizing Qini Coefficient and IRR," introduces a new causal stacking algorithm designed to optimize marketing strategies by accurately identifying the most persuadable customers.

Innovative Approach to Customer Engagement

By stacking machine learning models under causal CATE framework with novel optimization metrics , the research diverges from traditional meta-learner uplift methodologies. This novel approach allows for dynamic optimization of model weights through a neural network, effectively capturing complex interactions within datasets and significantly improving predictive accuracy and uplift scoring.

Our method not only increases the precision of identifying persuadable customers but also minimizes the risk of alienating potential customers by avoiding those less likely to respond," explained by the team.

Research and Development

The research was motivated by the observation that not all customers targeted with promotional offers during peak shopping periods, such as Black Friday, were motivated to make purchases. By refining uplift modeling techniques, the Duke team aims to enable companies like Starbucks to more accurately target and engage customers, potentially leading to increased sales and customer loyalty.

The model was trained and tested using a dataset involving a promotional campaign by Starbucks, where customers' purchasing behaviors were analyzed based on whether they received promotions. This data was then processed and utilized to train the model to differentiate between various types of buyers, improving the targeting of marketing campaigns.

Impressive Results and Future Applications

The results of the study were highly promising, with the ensemble model achieving a Qini index of 0.5843 and an Incremental Response Rate (IRR) of 0.4251. These metrics indicate a robust impact from the marketing strategies employed, with the ensemble model outperforming traditional methods in both predictive capabilities and campaign efficacy.

This research not only sets a new standard for uplift modeling but also promises more efficient allocation of marketing resources and a higher return on investment for promotional campaigns across various industries," said the team.

A Novel Method for Uplift Modeling with Weights Optimized by Maximizing Qini Coefficient and IRR in Starbucks Promotion

Zhankai, Ye
zy172
zhankai.ye@duke.edu

Yuhan(Skylar), Hou
yh383
yuhan.hou@duke.edu

Xinyan(Hathaway), Liu
xl441
hathaway.liu@duke.edu

Yujie(Johnny), Ye
yy413
yujie.ye@duke.edu

Hsuan-Chen(Justin), Kao
hk310
justinkao.44@duke.edu

1 Abstract

In this study, we introduce an advanced uplift modeling framework that leverages a stacked ensemble of diverse machine learning models. This framework represents a significant evolution from traditional uplift modeling techniques, which often rely on a single model and may not fully capture complex interactions within the dataset. Our proposed method computes the uplift, $\hat{\tau}(X)$, through the difference in conditional expectations:

$$\hat{\tau}(X) = \mathbb{E}[Y \mid X, A = 1] - \mathbb{E}[Y \mid X, A = 0],$$

where Y is the outcome variable, X represents the covariates, and A denotes the binary treatment assignment (1 for treated, 0 for control).

In a departure from initial methodologies that utilized average weighted determination, our model optimizes the weights $\omega_k(X)$ for each base model in the ensemble by maximizing the Incremental Response Rate (IRR) and Qini Index. This optimization ensures that:

$$\sum_{k=1}^{K} \omega_k(X) = 1, \quad \omega_k(X) \geq 0, \forall k.$$

The final uplift score, $\hat{\tau}_s(X)$, is then calculated as a weighted sum of the individual model predictions $\hat{\tau}_k(X)$, corresponding to the treatment effect:

$$\hat{\tau}_s(X) = \sum_{k=1}^{K} \omega_k(X)\hat{\tau}_k(X).$$

This method enhances the predictive precision and relevance of the uplift score, particularly in scenarios characterized by heterogeneous treatment effects, providing a robust mechanism for capturing the nuanced complexities within the data. Our approach not only improves the accuracy of uplift predictions but also ensures that the ensemble dynamically adapts to optimize key performance metrics across various data contexts.

2 Motivation

During the bustling shopping season of Black Friday in 2023, we noticed that some of our friends received promotional offers from Starbucks, yet these promotions did not motivate them to make purchases. This observation led us to realize that some targeted customers actually fit the 'sleeping dogs' category—individuals who might react negatively or remain indifferent to marketing efforts. This sparked our curiosity about the effectiveness of Starbucks' marketing strategy: Why target individuals who are unlikely to be persuaded?

With our background in statistics, we turned to uplift modeling to delve deeper into this issue. Our goal is to enhance the model's ability to distinguish between merely receptive customers and those who are truly persuadable. By refining this approach, we hope to enable Starbucks to more accurately identify and engage the persuadable customers, optimizing the impact of their promotional campaigns and avoiding the risk of alienating the 'sleeping dogs.' This would ensure that marketing efforts are not only more efficient but also more effective in increasing customer engagement and sales.

Figure 1: The Classic Uplift Segments Source: Towards Data Science.

3 Data preprocessing

3.1 Data Description

For this specific project, we choose the dataset about Starbucks, which is a dataset about an experiment involving a promotional campaign. As part of the experiment, some customers were given promotions to entice them to purchase a product. The dataset contains about customers' behaviors of whether purchase or not purchase, given promotion or not. The link of original dataset could be found in Starbucks Portfolio Excercise.

The whole dataset consist of 2 CSV files with the same columns, where the details of each table is presented in the following table:

Table	**Description**	**Feature**
training.csv, test.csv	raw samples	ID, Promotion, V1, V2, V3, V4, V5, V6, V7, purchase

Table 1: Description of data tables

Descriptions of fields could be find below. More could be find in the link of dataset:

- ID: Unique ID for customers
- Promotion: "Yes" for given promotion, "No" for not given promotion
- V1 - V7: customers' personal feature; feature names are not given due to customers' privacy
- purchase(int): 1 for purchase, 0 for not purchase

3.2 Data Preparation and Cleaning

Since the dataset is separated with training and test set, we want to combine them so that we could split our own train and test set. And essentially we want to investigate in two treatments: **User purchase or not purchase, promotion given or not**, we could do the following process:

1. Since the dataset is separated, we combine the train and test set together.
2. Since column "Promotion" has value "Yes" and "No", we encoded them to 1 and 0 respectively for future convenience.

3. Based on Figure 1 in Section 2, we could convert our dataset into 4 types of buyers with the following code:

```
combined_data['buyer_type'] = 0
combined_data.loc[(combined_data.Promotion == 0)&
(combined_data.purchase == 1), 'buyer_type'] = 1
combined_data.loc[(combined_data.Promotion == 1)&
(combined_data.purchase == 0), 'buyer_type'] = 2
combined_data.loc[(combined_data.Promotion == 0)&
(combined_data.purchase == 0), 'buyer_type'] = 3
```

   where the 4 types of buyers are:

   - Buyer 0: Persuadables — TR (Treatment and Response)
   - Buyer 1: Sure things — CR (Control and Response)
   - Buyer 2: Lost causes — TN (Treatment and No-response)
   - Buyer 3: CN (Control and No-response)

4. The numbers of 4 types are not balanced. Since the uplifting model requires sum of Buyer 0 and 2 to be around the same for the sum of Buyer 1 and 3, we could apply Up-sampling based on the highest number of 4 types (Here the highest among all is 62612):

```
from imblearn.over_sampling import SMOTE
features = combined_data.columns.difference(['buyer_type'])
X_train = combined_data[features]
Y_train = combined_data['buyer_type']

# Setting up SMOTE to balance the dataset
sm = SMOTE(sampling_strategy={0: 62612, 1: 62612, 2: 62612, 3: 62612},
    random_state=42)

# Apply SMOTE
X_train_upsamp, Y_train_upsamp = sm.fit_resample(X_train, Y_train)

# Convert the upsampled data back into DataFrame and Series
X_train_upsamp = pd.DataFrame(X_train_upsamp, columns=features)
Y_train_upsamp = pd.Series(Y_train_upsamp)
final_upsampled_data = pd.concat([X_train_upsamp, Y_train_upsamp], axis=1)
```

5. The final cleaned dataset could be found in the following Github repo: starbucks.csv

The new version of 4-way plot based on point 5) should look like the following:

Promotion 0		**Promotion 1**	
Purchase 1	**Purchase 0**	**Purchase 1**	**Purchase 0**
Sure Things (CR)	Sleeping Dogs (CN)	Persuadables (TR)	Lost Causes (TN)

Table 2: Figure 2: 4-way plot of Purchasing and Promotion

Note that above dataset might be abstract to understand. To easily understand the situation we want to explore, we could think of it in this way: You can easily think of a specific drink of Starbucks, say Pink Drink. And the customers are given a promotion of double amount of stars if they buy a drink. We classify the buyers into the 4 types we described above and we want to know that how these two treatments affects each other; and with the buyer's information, what insights could we gain about the campaign strategies of this product and what can we improve on the strategies.

4 Model Specification

Algorithm 1 Optimized Causal Stacking for Uplift Modeling

Input: Data set $\{(X_i, Y_i, Z_i)\}_{i=1}^{n}$, collection of CATE algorithms $\{\mathcal{A}_k\}_{k=1}^{K}$
Partition Data:
  Training set $\mathcal{S}_{\text{train}}$: $100(1-\alpha)\%$ of data
  Validation set $\mathcal{S}_{\text{val}}$: $100\alpha\%$ of data
Estimate Conditional Expectations:
for $t \in \{0, 1\}$ **do**
  Fit model $\hat{\mu}_t$ on $\mathcal{S}_{\text{train}}$: $\hat{\mu}_t = \text{Estimate } \mathbb{E}[Y_i \mid X_i, Z_i = t]$
end for
Fit CATE Algorithms:
for $k \in [K]$ **do**
  $\hat{\tau}_k \leftarrow \mathcal{A}_k(\mathcal{S}_{\text{val}} + \mathcal{S}_{\text{train}})$ with utilizing T-Learner
end for
Optimize Weights on Validation Set + Train Set:
  Define objective function $F(w)$ based on $\arg\max_{\omega_i, i=1,2,3,4}$ {Qini, IRR} on $\mathcal{S}_{\text{val}}$ + $\mathcal{S}_{\text{train}}$
  $w^* \leftarrow \text{argmax}_w F(w)$ subject to $w_k \geq 0$ and $\sum_{k=1}^{K} w_k = 1$
Compute Final Uplift Score Using the Entire Dataset:
  For each X in the dataset, compute:
  $\hat{\tau}_s(X) = \sum_{k=1}^{K} w_k^* \hat{\tau}_k(X)$
Evaluate Performance on Entire Dataset:
  Compute Qini and IRR of $\hat{\tau}_s$ using the entire dataset
Output: Dynamically optimized CATE function $\hat{\tau}_s(\cdot)$, final Qini index, and IRR

4.1 T-Learner

The T-Learner, or Two-model Learner, is a fundamental approach for estimating the Conditional Average Treatment Effect (CATE) by separately modeling the outcomes for the treatment and control groups using supervised learning techniques. The main idea is to fit two distinct predictive models: one for the treatment group and one for the control group.

Given a dataset $\{(X_i, Y_i, Z_i)\}_{i=1}^{n}$, where X_i represents covariates, Y_i the outcome, and Z_i the binary treatment indicator (with $Z_i = 1$ for treated and $Z_i = 0$ for control), the T-Learner approach is defined as follows:

- **Model for Treated:** Fit a model $\hat{\mu}_1$ to estimate $\mathbb{E}[Y \mid X, Z = 1]$ using only the observations from the treated group:
$$\hat{\mu}_1(X) = \arg\min_f \sum_{i:Z_i=1} L(Y_i, f(X_i)),$$
where L denotes a loss function, typically squared error for regression tasks.

- **Model for Control:** Fit a separate model $\hat{\mu}_0$ to estimate $\mathbb{E}[Y \mid X, Z = 0]$ using only the observations from the control group:
$$\hat{\mu}_0(X) = \arg\min_f \sum_{i:Z_i=0} L(Y_i, f(X_i)).$$

- **Estimation of CATE:** The CATE for a new observation with covariates X is estimated by the difference in predictions from the two models:
$$\hat{\tau}(X) = \hat{\mu}_1(X) - \hat{\mu}_0(X).$$

This method straightforwardly captures the differential effect of the treatment by comparing the outcomes predicted by the two models conditioned on the treatment assignment. The T-Learner is particularly effective when the interaction between the treatment and the covariates is assumed to be significant and when each model $\hat{\mu}_1$ and $\hat{\mu}_0$ is well-specified and accurately captures the conditional expectations within each group.

4.2 Machine Learning Algorithms

We then explore and incorporate a variety of Machine Learning algorithms, such as Random Forests and Neural Networks, into the famous Two-Model approach (T-learner), to estimate $\tau(X_i)$:

- *XGBoost T-learner*: This method fits two gradient boosting models $\hat{\mu}_1$ and $\hat{\mu}_0$ using data from treated and control units respectively. The fitted CATE function is $\hat{\tau}(x) = \hat{\mu}_1(x) - \hat{\mu}_0(x)$. All parameters were set at their default values in `XGBClassifier` function.

- *Random Forest T-learner*: Similar to *XGBoost Forest T-learner*, we used random forest models to fit the treated and control units to estimate $\hat{\tau}(x)$. All parameters were set at their default values in `RandomForestClassifier` function.

- *SVM T-learner*: Due to the limit of time and computing resources, SVM T-learn is excluded in the ensemble model.

- *Neural Network T-learner*: Though Neural Network is prevailing these days, we found that few academic paper incorporate Neural Network into Uplifting Model. We innovatively deploy two neural networks for treated and control units. Each group is modeled separately with neural networks that are trained, tuned via a random search for optimal hyperparameters, and evaluated using ROC curves and AUC metrics. The framework employs early stopping to prevent overfitting and uses the models to predict outcomes for new data, facilitating the estimation of conditional expectations under both treatment and control conditions, crucial for calculating treatment effects like the average treatment effect. This approach is valuable in scenarios with non-linear dependencies and complex interactions influencing outcomes.

4.3 Weighting Optimization Process for Maximizing final Qini Coefficient and Incremental Response Rate (IRR)

Objective: The aim is to optimize a vector of weights $\mathbf{w} = [w_1, w_2, w_3, w_4]$, which are applied to a set of conditional average treatment effect (CATE) estimations. The objective is to maximize the combination of the Incremental Response Rate (IRR) and Qini coefficient.

Objective Function: The function intended for maximization is defined as:

$$F(\mathbf{w}) = \text{IRR}(\mathbf{w}) + \text{Qini}(\mathbf{w})$$

For the minimization algorithm, the function is negated:

$$\min -F(\mathbf{w}) = -(\text{IRR}(\mathbf{w}) + \text{Qini}(\mathbf{w}))$$

Constraints:

- **Non-negativity:** Each weight w_k must be non-negative.

$$w_k \geq 0 \quad \forall k \in \{1, 2, 3, 4\}$$

- **Normalization:** The sum of all weights must equal one.

$$\sum_{k=1}^{4} w_k = 1$$

Decision Variables: The decision variables, $\mathbf{w}$, represent the weights assigned to each model's uplift score within the ensemble.

Optimization Setup: The optimization process includes defining the initial parameters and constraints:

- **Initial Guess:** Uniform distribution of weights, $\mathbf{w}^0 = [0.25, 0.25, 0.25, 0.25]$.
- **Bounds:** Each weight is bounded between 0 and 1, Bounds $= [(0, 1)] \times 4$.
- **Constraints:** Linear equality to ensure the weights sum to one.

Optimization Algorithm: Sequential Least Squares Programming (SLSQP) is selected for its suitability in handling nonlinear optimization problems with constraints:

- **Algorithm:** SLSQP (Sequential Least Squares Programming)
- **Objective:** Minimize $-F(\mathbf{w})$
- **Constraints:** Ensure normalization of weights through linear equality.

Execution: Using the `scipy.optimize.minimize` function:

```
result = minimize(calculate_combined_IRR, initial_weights,
                  method='SLSQP', bounds=bounds, constraints=cons)
```

Outcome
The output $\mathbf{w}^*$ provides the optimized weights for the ensemble, maximizing the combined metric of IRR and Qini, calculated as:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} -F(\mathbf{w})$$

subject to the specified bounds and constraints.

4.4 Results

Our results suggest that the ensemble algorithm outperforms the single algorithm overall. We evaluate the algorithm from two metrics:

- Incremental Response Rate (IRR): The Incremental Response Rate (IRR) is a metric that measures the additional percentage of a target group that responded to a campaign compared to a control group that was not exposed to the campaign.

  IRR > 0: A higher positive IRR suggests that a greater percentage of the target group took the desired action (such as making a purchase or signing up for a service) as a result of the campaign.

  IRR $= 0$: An IRR of zero suggests that the campaign had no noticeable effect on the behavior of the target group when compared to the control group.

  IRR < 0: A negative IRR indicates that the campaign potentially had an adverse effect, with a lower percentage of the target group taking the desired action compared to the control group.

- Qini Index: The Qini index is a metric used to evaluate the incremental impact of a treatment in a campaign by measuring the difference between the cumulative gains from treated and control groups over various threshold levels.

  Qini index > 0: This suggests that the treatment or campaign was effective and produced an incremental lift in the desired outcome. The higher the value, the greater the impact of the treatment on the treated group versus the control group.

  Qini index $= 0$: It implies that the treatment did not have any significant difference from the control group's behavior. There's no incremental lift attributable to the campaign.

  Qini index < 0: A negative Qini index would suggest that the treatment was counterproductive, leading to a worse outcome than if the treatment had not been administered at all. It could mean that the campaign had a negative effect on the desired outcome.

Model	Optimal Weight
Logistic Regression	0.25001355
XGBoost	0.24999748
Random Forest	0.24999235
Neural Network	0.24999662

Table 3: Optimal Weights for Different Models

Model	Qini Index	IRR
Logistic Regression	0.05324	0.1108
XGBoost	0.06702	-0.0196
Random Forest	-0.02864	0.4588
Neural Network	-0.04815	-0.0049
Ensemble Model	**0.5843**	**0.4251**

Table 4: Qini Index and IRR for Different Models

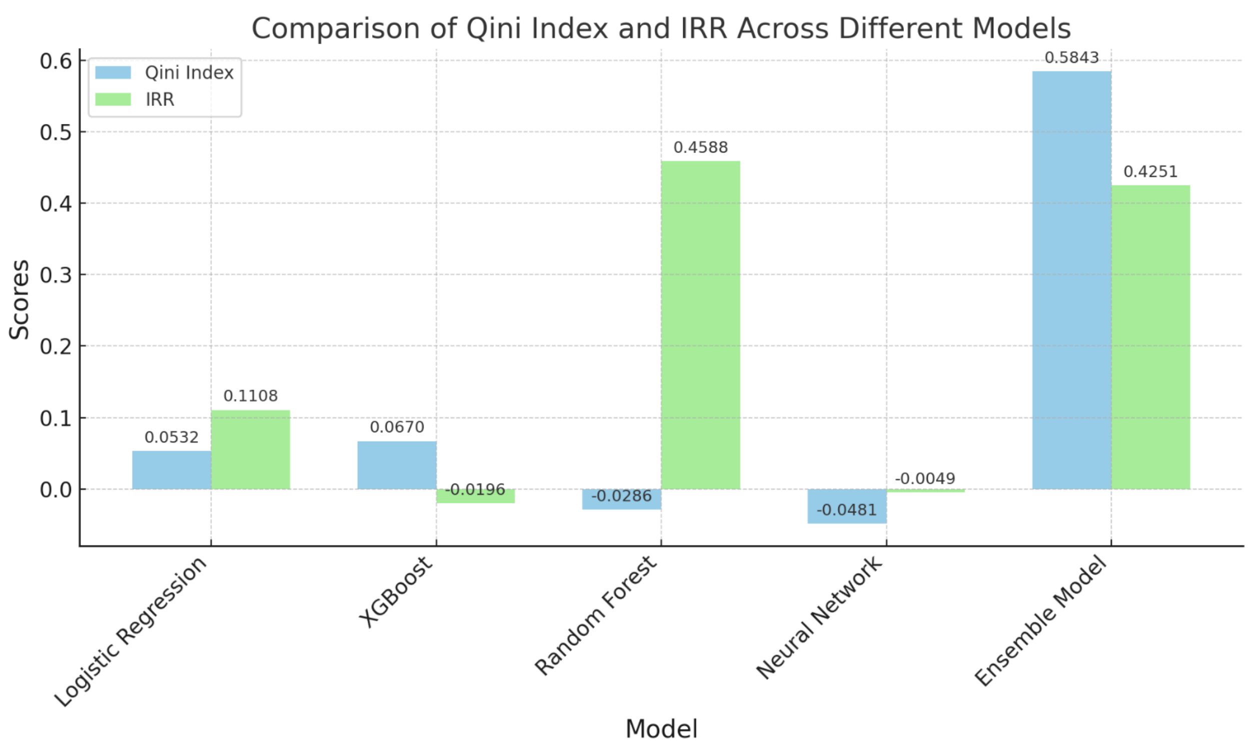


Figure 2: Model Comparison

Utilizing an optimization algorithm, we have determined the optimal weights for each predictive model to create an ensemble model tailored for our campaign **(Table 3)**. This ensemble approach has yielded impressive results **(Table 4)(Figure 2)**, notably achieving a **0.5843 Qini index**, which signifies a robust incremental impact from our marketing strategies. This high Qini index underscores the ensemble model's effectiveness in discerning the individuals whose decisions are most swayed by the campaign from those unaffected. In parallel, the ensemble model boasts a **0.4251 IRR**, illustrating its potent capacity to elicit a markedly higher proportion of positive responses from the target demographic in response to the campaign, relative to a baseline established by a control group. These figures not only highlight the ensemble model's advanced predictive capabilities but also underscore its considerable utility in enhancing campaign efficacy.

5 Conclusion

In conclusion, our investigation into the optimization of ensemble modeling for uplift campaigns has been a resounding success, as evidenced by the significant Qini index and IRR values attained. The optimal combination of machine learning models, as reflected in our ensemble, has proven its mettle by not only discerning the nuanced dynamics of customer responsiveness but also by substantially amplifying the positive outcomes of the campaign. The Qini index of 0.5843 and IRR of 0.4251 are testaments to the model's effectiveness in identifying and influencing the persuadable segment of customers. Our research contributes a novel and effective approach to uplift modeling, promising a more efficient allocation of marketing resources and a higher return on investment for promotional campaigns, thereby setting a new standard for future studies in this domain.

While our ensemble model has demonstrated substantial efficacy in uplift campaigns, it is not without its limitations. A constraint is the absence of demographic information within our dataset, which restricts our capacity to finely segment the customer base and provide detailed insights into the demographics of the target audience. This limitation could potentially hinder the customization and personalization aspects of marketing strategies, which are increasingly vital in contemporary campaigns. Furthermore, the generalizability of our results may be impacted by the specificity of the dataset to a particular brand and promotional context, calling for further validation across diverse industries and campaign types. Despite these constraints, our research offers valuable advancements in uplift modeling techniques and establishes a foundation for future investigations to build upon, with an emphasis on integrating broader datasets and refining model applicability across varied marketing landscapes.

6 FAQ

1. Compared to a standard uplifting model, what are the innovative aspects of your model?

   Answer: The model utilizes a stacked ensemble of diverse machine learning models, moving beyond conventional uplift methodologies that typically rely on a single predictive model. This approach enhances the ability to capture complex interactions within the dataset, providing a more robust and accurate prediction of the uplift. Dynamic Weight Optimization: Unlike static model weighting methods, the proposed model employs a neural network to dynamically optimize the weights for each constituent model within the ensemble. This neural network uses the covariates (X) as inputs to output distinct weights for each base model, ensuring that the influence of each model is adjusted based on the specific context of the input data. This flexibility allows for more precise adaptation to varying data scenarios, potentially improving the predictive accuracy and relevance of the uplift scores. Focus on Heterogeneous Treatment Effects: The architecture is specifically designed to handle scenarios characterized by heterogeneous treatment effects more effectively. By dynamically adjusting the contribution of each model in the ensemble based on the data it receives, the framework can more accurately identify and respond to different patterns of responsiveness to the treatment among different groups or situations. These innovative features aim to significantly enhance the performance of uplift modeling by providing a more nuanced, flexible, and accurate approach to estimating the incremental impact of treatments or interventions, making it particularly suitable for complex and varied datasets like those often encountered in real-world marketing scenarios.

2. How much incremental sales does the promotion bring to each customer?

   Answer: The incremental sales can be calculated but it will be based on the Net Incremental Revenue (NIR) and it requires revenue generated by both the treatment and control groups, along with the total costs of the treatment. This allows you to assess the profitability of a marketing campaign by comparing the net revenue generated against the costs, determining if the financial outcomes justify the expenditures. However, we don't have relevant features in our dataset. Therefore, we cannot get the concrete value about incremental sales.

3. If you can only send promotions to the top 10,000 customers, which customers would receive the promotion? Why?

   Answer: Based on our results, we will utilize our best model and select the 10,000 users with the highest uplifting scores to send promotions. We can maximum our promotion effect by choosing them to send promotions since an uplifting score in a campaign measures how much more likely individuals exposed to the campaign are to take a desired action, in our case the desired action is purchasing products, compared to those who were not exposed.

4. If you must remove 10,000 customers from those who received the promotion, which customers would you exclude from the campaign? Why?

   Answer: Based on our results, we will use our best model and select the 10,000 users with the lowest uplifting scores to send promotions. Because they are the least likely to choose to buy the product due to promotions.

5. Based on your final results, which audience would you target for the promotion? What data do you have to support your recommendations?

   Answer: According to our final conclusion, users having positive uplifting scores will be our target for the promotion since they are likely to buy products because of our promotions. Furthermore, people can try to get some demographic information about our target customers based on our model, but unfortunately, we don't have enough features to acquire these details.

6. Where does the 4 buyer types go? How does those 4 types of buyer related to your result?

   Answer: The 4 types of buyer are initially setting up based on the definition of Uplifting model, which is our original "guess" about how persuadable customers are. The buyer types are dropped during model fitting for prediction purpose. But the final Qini Index and IRR could actually refer back to our initial buyer types, which we could compare them and see how well is our initial category and who are those customers that are actually persuadable despite the types we gave them.

7 Contribution

This project is the culmination of our collective efforts, forged through one and a half weeks of intensive in-person collaboration.

Figure 3: Meme Source: 32 Funny Teamwork Memes.

References

[1] Kevin Wu. *Ensemble Method for Estimating Individualized Treatment*, Mar. 2022, `http://kevinwhan.github.io/files/paper-ensemble.pdf`.

[2] M. Sołtys, S. Jaroszewicz, and P. Rzepakowski. "Ensemble methods for uplift modeling," *Data Mining and Knowledge Discovery*, 29(6), 2015, pp. 1531–1559.

[3] Jannik Rößler, Roman Tilly, and Detlef Schoder. "To Treat, or Not to Treat: Reducing Volatility in Uplift Modeling Through Weighted Ensembles." In *Proceedings of the 54th Hawaii International Conference on System Sciences (HICSS)*, 2021, DOI: 10.24251/HICSS.2021.193.

In [ ]:
```
# @title Package needed
!pip install xgboost
!pip install scikit-uplift
!pip install imblearn

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from google.colab import files
```

```
Requirement already satisfied: xgboost in /usr/local/lib/python3.10/dist-packages (2.0.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from xgboost) (1.25.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from xgboost) (1.11.4)
Requirement already satisfied: scikit-uplift in /usr/local/lib/python3.10/dist-packages (0.5.1)
Requirement already satisfied: scikit-learn>=0.21.0 in /usr/local/lib/python3.10/dist-packages (from scikit-uplift) (1.2.2)
Requirement already satisfied: numpy>=1.16 in /usr/local/lib/python3.10/dist-packages (from scikit-uplift) (1.25.2)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from scikit-uplift) (2.0.3)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (from scikit-uplift) (3.7.1)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from scikit-uplift) (2.31.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from scikit-uplift) (4.66.2)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.21.0->scikit-uplift) (1.11.4)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.21.0->scikit-uplift) (1.4.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.21.0->scikit-uplift) (3.4.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-uplift) (1.2.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-uplift) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-uplift) (4.51.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-uplift) (1.4.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-uplift) (24.0)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-uplift) (10.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-uplift) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-uplift) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas->scikit-uplift) (2024.1)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas->scikit-uplift) (2024.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->scikit-uplift) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->scikit-uplift) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->scikit-uplift) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->scikit-uplift) (2024.2.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib->scikit-uplift) (1.16.0)
```

In [ ]:
```
# @title Data Processing
# Load the training and test datasets
training_data = pd.read_csv('https://raw.githubusercontent.com/joshxinjie/Data_Scientist_Nanodegree/master/starbucks_portfolio_exercise/training.cs
test_data = pd.read_csv('https://raw.githubusercontent.com/joshxinjie/Data_Scientist_Nanodegree/master/starbucks_portfolio_exercise/Test.csv')

# Combine the datasets
combined_data = pd.concat([training_data, test_data], ignore_index=True)
combined_data['Promotion'] = combined_data['Promotion'].map({'Yes': 1, 'No': 0})
print(combined_data.head(10))
```

In [ ]:
```
# Count unique value of 'purchase'
combined_data["purchase"].value_counts()
```

In [ ]:
```
# Convert customers to 4 buyer types
combined_data['buyer_type'] = 0
combined_data.loc[(combined_data.Promotion == 0)&(combined_data.purchase == 1), 'buyer_type'] = 1
combined_data.loc[(combined_data.Promotion == 1)&(combined_data.purchase == 0), 'buyer_type'] = 2
combined_data.loc[(combined_data.Promotion == 0)&(combined_data.purchase == 0), 'buyer_type'] = 3
combined_data['buyer_type'].value_counts()
```

In [ ]:
```
# Perform up-sampling to balance the data
features = combined_data.columns.difference(['buyer_type'])
X_train = combined_data[features]
Y_train = combined_data['buyer_type']

# Setting up SMOTE to balance the dataset
sm = SMOTE(sampling_strategy={0: 62612, 1: 62612, 2: 62612, 3: 62612}, random_state=42)

# Apply SMOTE
X_train_upsamp, Y_train_upsamp = sm.fit_resample(X_train, Y_train)

# Convert the upsampled data back into DataFrame and Series
X_train_upsamp = pd.DataFrame(X_train_upsamp, columns=features)
Y_train_upsamp = pd.Series(Y_train_upsamp)
final_upsampled_data = pd.concat([X_train_upsamp, Y_train_upsamp], axis=1)
final_upsampled_data.head(10)
```

In [ ]:
```
# check the final data see if it's balanced
final_upsampled_data['Promotion'].value_counts()
final_upsampled_data['purchase'].value_counts()
```

In [ ]:
```
# Download the final dataset and put it in Github repo
final_upsampled_data.to_csv('starbuck.csv', index=False)
files.download('starbuck.csv')
```

In [ ]:
```
# @title Load Data
data = pd.read_csv("https://raw.githubusercontent.com/STA561-Final-Project/causal-ml/main/starbuck.csv")

# Display the first few rows of the transformed data
data
```

Out[ ]:

	ID	Promotion	V1	V2	V3	V4	V5	V6	V7	purchase	buyer_type
0	1	0	2	30.443518	-1.165083	1	1	3	2	0	3
1	3	0	3	32.159350	-0.645617	2	3	2	2	0	3
2	4	0	2	30.431659	0.133583	1	1	4	2	0	3
3	5	0	0	26.588914	-0.212728	2	1	4	2	0	3
4	8	1	3	28.044331	-0.385883	1	1	2	2	0	2
...	...	...	...	...	...	...	...	...	...	...	...
250443	123137	1	2	22.492320	0.795937	2	2	1	1	0	2
250444	50821	1	1	29.660005	1.264919	2	2	1	1	0	2
250445	33347	1	0	28.862809	0.963575	2	3	2	1	0	2
250446	16269	1	1	35.587445	0.592060	1	2	3	1	0	2
250447	71561	1	2	35.107513	1.511598	2	1	3	2	0	2

250448 rows × 11 columns

In [ ]:
```python
# Filter out the columns 'user', 'cate_id', 'customer'
filtered_data = data.drop(['ID', 'buyer_type'], axis=1)

# Display the first few rows of the filtered data
filtered_data
# This will print distinct values for each column
for column in filtered_data.columns:
    print(f"Distinct values in '{column}': {filtered_data[column].unique()}")
```

```
Distinct values in 'Promotion': [0 1]
Distinct values in 'V1': [2 3 0 1]
Distinct values in 'V2': [30.4435178  32.1593501  30.4316591  ... 28.86280921 35.58744477
 35.10751318]
Distinct values in 'V3': [-1.1650834  -0.6456167   0.13358341 ...  0.9635752   0.59206016
  1.51159774]
Distinct values in 'V4': [1 2]
Distinct values in 'V5': [1 3 4 2]
Distinct values in 'V6': [3 2 4 1]
Distinct values in 'V7': [2 1]
Distinct values in 'purchase': [0 1]
```

In [ ]:
```python
# Columns to encode
columns_to_encode = ['V1', 'V4', 'V5', 'V6', 'V7']

# Apply one-hot encoding
cleaned_data = pd.get_dummies(filtered_data, columns=columns_to_encode, drop_first=False)

# Show new DataFrame structure
print(cleaned_data.head())
```

```
   Promotion         V2        V3  purchase   V1_0   V1_1   V1_2   V1_3  \
0          0  30.443518 -1.165083         0  False  False   True  False
1          0  32.159350 -0.645617         0  False  False  False   True
2          0  30.431659  0.133583         0  False  False   True  False
3          0  26.588914 -0.212728         0   True  False  False  False
4          1  28.044331 -0.385883         0  False  False  False   True

    V4_1   V4_2   V5_1   V5_2   V5_3   V5_4   V6_1   V6_2   V6_3   V6_4  \
0   True  False   True  False  False  False  False  False   True  False
1  False   True  False  False   True  False  False   True  False  False
2   True  False   True  False  False  False  False  False  False   True
3  False   True   True  False  False  False  False  False  False   True
4   True  False   True  False  False  False  False   True  False  False

    V7_1  V7_2
0  False  True
1  False  True
2  False  True
3  False  True
4  False  True
```

In [ ]:
```python
cleaned_data
```

Out[ ]:

	Promotion	V2	V3	purchase	V1_0	V1_1	V1_2	V1_3	V4_1	V4_2	V5_1	V5_2	V5_3	V5_4	V6_1	V6_2	V6_3	V6_4	V7_1	V7_2
0	0	30.443518	-1.165083	0	False	False	True	False	True	False	True	False	False	False	False	False	True	False	False	True
1	0	32.159350	-0.645617	0	False	False	False	True	False	True	False	False	True	False	False	True	False	False	False	True
2	0	30.431659	0.133583	0	False	False	True	False	True	False	True	False	False	False	False	False	False	True	False	True
3	0	26.588914	-0.212728	0	True	False	False	False	False	True	True	False	False	False	False	False	False	True	False	True
4	1	28.044331	-0.385883	0	False	False	False	True	True	False	True	False	False	False	False	True	False	False	False	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
250443	1	22.492320	0.795937	0	False	False	True	False	False	True	False	True	False	False	True	False	False	False	True	False
250444	1	29.660005	1.264919	0	False	True	False	False	False	True	False	True	False	False	True	False	False	False	True	False
250445	1	28.862809	0.963575	0	True	False	False	False	False	True	False	False	True	False	False	True	False	False	True	False
250446	1	35.587445	0.592060	0	False	True	False	False	True	False	False	True	False	False	False	False	True	False	True	False
250447	1	35.107513	1.511598	0	False	False	True	False	False	True	True	False	False	False	False	False	True	False	False	True

250448 rows × 20 columns

In [ ]:
```python
# Converting all boolean columns to integer (1 for True, 0 for False), except 'price'
for column in cleaned_data.columns:
    if cleaned_data[column].dtype == bool and column != 'price':
```

```
        cleaned_data[column] = cleaned_data[column].astype(int)

cleaned_data
```

Out[ ]:

	Promotion	V2	V3	purchase	V1_0	V1_1	V1_2	V1_3	V4_1	V4_2	V5_1	V5_2	V5_3	V5_4	V6_1	V6_2	V6_3	V6_4	V7_1	V7_2
0	0	30.443518	-1.165083	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
1	0	32.159350	-0.645617	0	0	0	0	1	0	1	0	0	1	0	0	1	0	0	0	1
2	0	30.431659	0.133583	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	0	1
3	0	26.588914	-0.212728	0	1	0	0	0	0	1	1	0	0	0	0	0	0	1	0	1
4	1	28.044331	-0.385883	0	0	0	0	1	1	0	1	0	0	0	0	1	0	0	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
250443	1	22.492320	0.795937	0	0	0	1	0	0	1	0	1	0	0	1	0	0	0	1	0
250444	1	29.660005	1.264919	0	0	1	0	0	0	1	0	1	0	0	1	0	0	0	1	0
250445	1	28.862809	0.963575	0	1	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0
250446	1	35.587445	0.592060	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0	1	0
250447	1	35.107513	1.511598	0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	0	1

250448 rows × 20 columns

In [ ]:

```
# Calculate the counts
purchase_counts_with_promotion = cleaned_data[cleaned_data['Promotion'] == 1]['purchase'].value_counts()
purchase_counts_without_promotion = cleaned_data[cleaned_data['Promotion'] == 0]['purchase'].value_counts()

# Output the results
print("With promotion:")
print(purchase_counts_with_promotion)

print("\nWithout promotion:")
print(purchase_counts_without_promotion)
```

```
With promotion:
purchase
0    62612
1    62612
Name: count, dtype: int64

Without promotion:
purchase
0    62612
1    62612
Name: count, dtype: int64
```

In [ ]:

```
# @title Data Splitting
## We would random split all the data into S_training and S_avg

S_train, S_test = train_test_split(cleaned_data, test_size=0.2, random_state=42)

S_train_campaign_x = S_train[S_train['Promotion'] == 1].drop('purchase',axis=1)
S_train_campaign_y = S_train[S_train['Promotion'] == 1]['purchase']

S_test_campaign_x = S_test[S_test['Promotion'] == 1].drop('purchase',axis=1)
S_test_campaign_y = S_test[S_test['Promotion'] == 1]['purchase']

S_train_nocam_x = S_train[S_train['Promotion'] == 0].drop('purchase',axis=1)
S_train_nocam_y = S_train[S_train['Promotion'] == 0]['purchase']

S_test_nocam_x = S_test[S_test['Promotion'] == 0].drop('purchase',axis=1)
S_test_nocam_y = S_test[S_test['Promotion'] == 0]['purchase']
```

In [ ]:

```
S_train
```

Out[ ]:

	Promotion	V2	V3	purchase	V1_0	V1_1	V1_2	V1_3	V4_1	V4_2	V5_1	V5_2	V5_3	V5_4	V6_1	V6_2	V6_3	V6_4	V7_1	V7_2
124794	0	27.195293	-1.078506	0	0	0	1	0	0	1	0	1	0	0	1	0	0	0	1	0
50595	1	25.778686	0.047006	0	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1
17488	1	26.137898	0.826206	0	0	0	1	0	0	1	0	1	0	0	0	1	0	0	0	1
26933	0	30.152994	0.133583	0	0	0	1	0	1	0	0	0	1	0	0	0	0	1	0	1
206618	0	32.040431	0.671517	1	1	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
119879	0	34.840657	0.133583	0	1	0	0	0	0	1	0	1	0	0	1	0	0	0	0	1
103694	1	28.968405	-0.472461	0	0	0	1	0	0	1	0	1	0	0	0	1	0	0	1	0
131932	1	30.450284	-1.360190	1	0	1	0	0	0	1	0	1	0	0	1	0	0	0	0	1
146867	1	30.591247	-0.298434	1	0	1	0	0	0	1	0	1	0	0	1	0	0	0	1	0
121958	0	31.586639	-1.338239	0	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	1

200358 rows × 20 columns

In [ ]:

```
S_test
```

Out[ ]:

	Promotion	V2	V3	purchase	V1_0	V1_1	V1_2	V1_3	V4_1	V4_2	V5_1	V5_2	V5_3	V5_4	V6_1	V6_2	V6_3	V6_4	V7_1	V7_2
197377	0	26.369153	-0.230870	1	0	0	1	0	1	0	0	1	0	0	1	0	0	0	1	0
203948	0	31.616423	1.598552	1	0	0	1	0	0	1	0	1	0	0	0	0	1	0	0	1
65189	0	35.782936	0.393317	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	1
71617	0	24.319409	1.605406	0	0	0	1	0	0	1	0	1	0	0	0	1	0	0	1	0
13678	0	33.102113	-1.338239	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
149733	1	37.476986	0.061571	1	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	1
243004	0	32.043784	0.249535	1	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0
235689	0	28.832741	0.482255	1	1	0	0	0	1	0	0	0	1	0	0	0	1	0	1	0
221809	0	32.244938	-0.107131	1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	0
148322	1	35.115643	0.145839	1	0	1	0	0	0	1	0	1	0	0	0	1	0	0	1	0

50090 rows × 20 columns

Modeling

In [ ]:

```
# @title Base Logistic Model
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix


# Initialize the logistic regression model
logistic_model = LogisticRegression()

# Train the logistic regression model for customers with compaign
camp_model = logistic_model.fit(S_train_campaign_x, S_train_campaign_y)
predictions = camp_model.predict(S_test_campaign_x)
accuracy = accuracy_score(S_test_campaign_y, predictions)
conf_matrix = confusion_matrix(S_test_campaign_y, predictions)

prob_cam = logistic_model.predict_proba(S_test_campaign_x)

print(f"Accuracy: {accuracy}")
print("Confusion Matrix:")
print(conf_matrix)

# Train the logistic regression model for customers without compaign
logistic_model = LogisticRegression()

# Train the logistic regression model for customers with compaign
nocam_model = logistic_model.fit(S_train_nocam_x, S_train_nocam_y)
predictions_nocam = nocam_model.predict(S_test_nocam_x)
accuracy_nocam = accuracy_score(S_test_nocam_y, predictions_nocam)
conf_matrix_nocam = confusion_matrix(S_test_nocam_y, predictions_nocam)

print(f"Accuracy: {accuracy_nocam}")
print("Confusion Matrix:")
print(conf_matrix_nocam)

S_whole_x = cleaned_data.drop('purchase',axis=1)

# Calculate uplifting scores
prob_cam = camp_model.predict_proba(S_whole_x)
highest_prob_camp_log = np.max(prob_cam, axis=1)

prob_nocam = nocam_model.predict_proba(S_whole_x)
highest_prob_nocamp_log = np.max(prob_nocam, axis=1)


uplift_scores_log = highest_prob_camp_log - highest_prob_nocamp_log
print(uplift_scores_log)
```

```
Accuracy: 0.6863829956613462
Confusion Matrix:
[[7977 4632]
 [3247 9267]]
Accuracy: 0.7153442544158289
Confusion Matrix:
[[8061 4232]
 [2875 9799]]
[-0.05972508 -0.07800484  0.01961386 ...  0.09444583 -0.25173221
 -0.08953818]
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

XGBoost T learner

In [ ]:

```
from sklearn.model_selection import GridSearchCV
from xgboost import XGBClassifier
```

```python
# Define the parameter grid to search
param_grid = {
    'max_depth': [3, 4, 5],
    'min_child_weight': [1, 5, 10],
    'gamma': [0.5, 1, 1.5],
    'subsample': [0.6, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.8, 1.0],
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.1, 0.2]
}

# Initialize the XGBoost classifier
xgb_clf = XGBClassifier(use_label_encoder=False, eval_metric='logloss')

# Initialize the Grid Search model
grid_search = GridSearchCV(xgb_clf, param_grid, scoring='roc_auc', cv=3, verbose=2)
grid_search_nocam = GridSearchCV(xgb_clf, param_grid, scoring='roc_auc', cv=3, verbose=2)

# Fit the grid search to the data
grid_search.fit(S_train_campaign_x, S_train_campaign_y)
grid_search_nocam.fit(S_train_nocam_x, S_train_nocam_y)

# Get the best parameters
best_parameters = grid_search.best_params_
best_parameters_nocam = grid_search_nocam.best_params_
print("Best Parameters:", best_parameters)
print("Best Parameters:", best_parameters_nocam)
```

In [ ]:
```python
import numpy as np
import pandas as pd
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, confusion_matrix


best_parameters_cam = {'colsample_bytree': 0.8, 'gamma': 0.5, 'learning_rate': 0.2, 'max_depth': 4, 'min_child_weight': 1, 'n_estimators': 100, 'su
best_parameters_nocam = {'colsample_bytree': 0.6, 'gamma': 1.5, 'learning_rate': 0.01, 'max_depth': 3, 'min_child_weight': 5, 'n_estimators': 100,


# Initialize the XGBoost models
camp_model_xgb = XGBClassifier(use_label_encoder=False, eval_metric='logloss')
nocam_model_xgb = XGBClassifier(use_label_encoder=False, eval_metric='logloss')

# Train the XGBoost model for customers with campaign
camp_model_xgb.fit(S_train_campaign_x, S_train_campaign_y)
predictions_camp = camp_model_xgb.predict(S_test_campaign_x)
accuracy_camp = accuracy_score(S_test_campaign_y, predictions_camp)
conf_matrix_camp = confusion_matrix(S_test_campaign_y, predictions_camp)

print(f"Accuracy with campaign: {accuracy_camp}")
print("Confusion Matrix with campaign:")
print(conf_matrix_camp)

# Train the XGBoost model for customers without campaign
nocam_model_xgb.fit(S_train_nocam_x, S_train_nocam_y)
predictions_nocam = nocam_model_xgb.predict(S_test_nocam_x)
accuracy_nocam = accuracy_score(S_test_nocam_y, predictions_nocam)
conf_matrix_nocam = confusion_matrix(S_test_nocam_y, predictions_nocam)

print(f"Accuracy without campaign: {accuracy_nocam}")
print("Confusion Matrix without campaign:")
print(conf_matrix_nocam)

# Calculate uplifting scores using the trained models on a common dataset for predictions
prob_camp_xgb = camp_model_xgb.predict_proba(S_whole_x)
# Select the highest probability between the two predicted classes for each sample
highest_prob_camp = np.max(prob_camp_xgb, axis=1)

prob_nocam_xgb = nocam_model_xgb.predict_proba(S_whole_x)  # Only take the probability of positive class
highest_prob_nocam = np.max(prob_nocam_xgb, axis=1)

# Calculate uplift scores
uplift_scores_xgb = highest_prob_camp - highest_prob_nocam
print("Uplift Scores (XGBoost):")
print(uplift_scores_xgb)
```

```
Accuracy with campaign: 0.9004497870477252
Confusion Matrix with campaign:
[[12106   503]
 [ 1998 10516]]
Accuracy without campaign: 0.9145271758721513
Confusion Matrix without campaign:
[[11790   503]
 [ 1631 11043]]
Uplift Scores (XGBoost):
[ 0.12810302 -0.04620218 -0.07439268 ... -0.00447685 -0.05344957
 -0.10948408]
```

Random Forest T Learner

In [ ]:
```python
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix

# Initialize Random Forest classifiers for treatment and control groups
rf_treatment = RandomForestClassifier(n_estimators=100, random_state=42)
rf_control = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the classifiers on respective group data
rf_treatment.fit(S_train_campaign_x, S_train_campaign_y)  # Treatment group training data
rf_control.fit(S_train_nocam_x, S_train_nocam_y)          # Control group training data
```

```
# Predict probabilities on a common dataset (S_whole_x) for both models
prob_treatment = rf_treatment.predict_proba(S_whole_x)[:, 1]  # Probabilities for the positive class
prob_control = rf_control.predict_proba(S_whole_x)[:, 1]      # Probabilities for the positive class

# Calculate uplift scores by subtracting control probabilities from treatment probabilities
uplift_scores_rf = prob_treatment - prob_control

# Print results
print(f"Accuracy of Random Forest: {accuracy_rf}")
print("Uplift Scores (Random Forest):")
print(uplift_scores_rf)
```

```
Accuracy of Random Forest: 0.850017911873582
Uplift Scores (Random Forest):
[ 0.18  0.    0.47 ...  0.   -0.51  0.03]
```

Neural Network

In [ ]:
```
# Library
!pip install keras-tuner
import pandas as pd
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import make_column_transformer
from sklearn.model_selection import GroupShuffleSplit

from tensorflow import keras
from tensorflow.keras import layers
from sklearn.metrics import roc_curve, roc_auc_score
from keras_tuner import RandomSearch
from kerastuner.engine.hyperparameters import HyperParameters
```

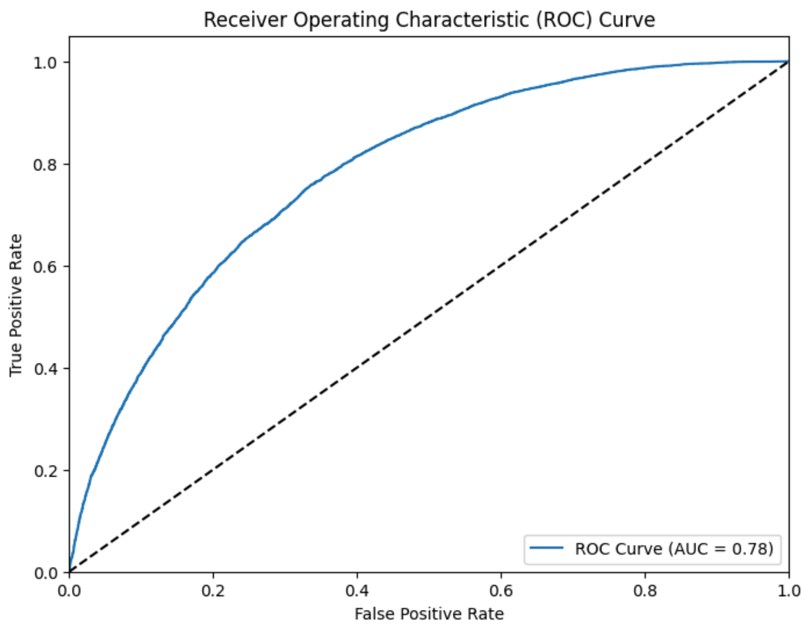
```
Requirement already satisfied: keras-tuner in /usr/local/lib/python3.10/dist-packages (1.4.7)
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (2.15.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (24.0)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (2.31.0)
Requirement already satisfied: kt-legacy in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (1.0.5)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (2024.2.2)
```

Campaign NN model

In [ ]:
```
# NN with Campaign

features_num = ['V2', 'V3']
features_cat = ['V1_0', 'V1_1', 'V1_2', 'V1_3',
       'V4_1', 'V4_2', 'V5_1', 'V5_2', 'V5_3', 'V5_4', 'V6_1', 'V6_2', 'V6_3',
       'V6_4', 'V7_1', 'V7_2']


X_train = S_train_campaign_x.drop(columns=['Promotion'])
X_valid = S_test_campaign_x.drop(columns=['Promotion'])
y_train = S_train_campaign_y
y_valid = S_test_campaign_y


input_shape = (X_train.shape[1],)  # Creating a tuple with a single element

# Building the deep learning model
model = keras.Sequential([
    layers.Dense(units=64, activation='relu', input_shape= input_shape),
    layers.Dense(units=64, activation='relu'),
    layers.Dense(units=1, activation='sigmoid')
])

# Compiling the model
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['binary_accuracy']
)

# This will run the model and plot the learning curve
early_stopping = keras.callbacks.EarlyStopping(
    patience=5,
    min_delta=0.001,
    restore_best_weights=True,
)

history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=512,
    epochs=200,
    callbacks=[early_stopping],

)

y_pred = model.predict(X_valid)

# Calculate the ROC curve
fpr, tpr, thresholds = roc_curve(y_valid, y_pred)
roc_auc = roc_auc_score(y_valid, y_pred)
# Plot the ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
```

```
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

```
Epoch 1/200
196/196 [==============================] - 1s 4ms/step - loss: 0.6050 - binary_accuracy: 0.6665 - val_loss: 0.5730 - val_binary_accuracy: 0.6969
Epoch 2/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5686 - binary_accuracy: 0.6980 - val_loss: 0.5705 - val_binary_accuracy: 0.7004
Epoch 3/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5633 - binary_accuracy: 0.7041 - val_loss: 0.5622 - val_binary_accuracy: 0.7062
Epoch 4/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5622 - binary_accuracy: 0.7041 - val_loss: 0.5605 - val_binary_accuracy: 0.7050
Epoch 5/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5602 - binary_accuracy: 0.7077 - val_loss: 0.5598 - val_binary_accuracy: 0.7080
Epoch 6/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5582 - binary_accuracy: 0.7073 - val_loss: 0.5575 - val_binary_accuracy: 0.7097
Epoch 7/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5580 - binary_accuracy: 0.7078 - val_loss: 0.5672 - val_binary_accuracy: 0.7015
Epoch 8/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5584 - binary_accuracy: 0.7075 - val_loss: 0.5613 - val_binary_accuracy: 0.7038
Epoch 9/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5562 - binary_accuracy: 0.7088 - val_loss: 0.5568 - val_binary_accuracy: 0.7090
Epoch 10/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5560 - binary_accuracy: 0.7094 - val_loss: 0.5585 - val_binary_accuracy: 0.7060
Epoch 11/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5550 - binary_accuracy: 0.7094 - val_loss: 0.5625 - val_binary_accuracy: 0.7045
786/786 [==============================] - 1s 1ms/step
```



Campaign NN model - Tuning

In [ ]:

```
input_shape = (X_train.shape[1])  # Creating a tuple with a single element

# Define a function that builds your Keras model with hyperparameters
def build_model(hp):
    model = keras.Sequential()
    model.add(layers.Dense(units=hp.Int('units', min_value=32, max_value=512, step=32), activation='relu', input_shape=(input_shape,)))
    model.add(layers.Dense(1, activation='sigmoid'))  # Output layer for binary classification
    model.compile(
        optimizer=hp.Choice('optimizer', values=['adam', 'rmsprop', 'sgd']),
        loss='binary_crossentropy',
        metrics=['binary_accuracy']
    )
    return model

tuner = RandomSearch(
    build_model,
    objective='binary_accuracy',
    max_trials=5,  # Number of different hyperparameter combinations to try
    directory='my_dir_cam',  # Directory where logs and results will be stored
    project_name='my_project_cam'  # Name for the tuning project
)

tuner.search(X_train, y_train, epochs=10, validation_data=(X_valid, y_valid))

best_model = tuner.get_best_models(num_models=1)[0]
best_hyperparameters = tuner.get_best_hyperparameters(num_trials=1)[0]
best_model.summary()
```

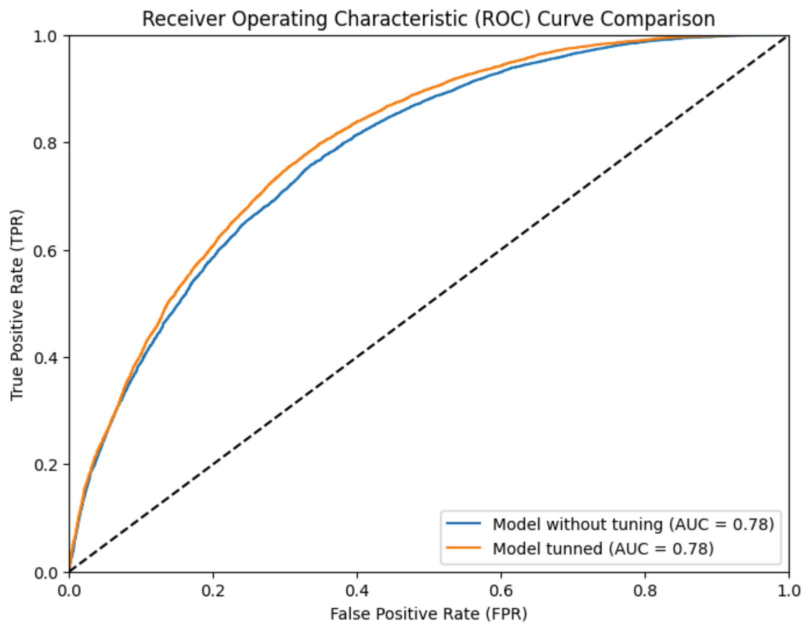
```
Reloading Tuner from my_dir_cam/my_project_cam/tuner0.json
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense (Dense)               (None, 480)               9120      
                                                                 
 dense_1 (Dense)             (None, 1)                 481       
                                                                 
=================================================================
Total params: 9601 (37.50 KB)
Trainable params: 9601 (37.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [ ]:

```python
# Build the best model using the best hyperparameters
best_model = build_model(best_hyperparameters)

# Train the best model with your training data
history_tuned = best_model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=512,
    epochs=200,
    callbacks=[early_stopping],
)

y_pred_tuned = best_model.predict(X_valid)

# Calculate the ROC curve
fpr2, tpr2, thresholds = roc_curve(y_valid, y_pred_tuned)

# Calculate the ROC AUC score
roc_auc_tuned = roc_auc_score(y_valid, y_pred)

print("ROC AUC Score:", roc_auc_tuned)

# Print both models to see if it has really improved
# Create a plot to compare the ROC curves
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'Model without tuning (AUC = {roc_auc:.2f})')
plt.plot(fpr2, tpr2, label=f'Model tunned (AUC = {roc_auc_tuned:.2f})')
plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line for reference
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Receiver Operating Characteristic (ROC) Curve Comparison')
plt.legend(loc='lower right')
plt.show()
```

```
WARNING:tensorflow:Detecting that an object or model or tf.train.Checkpoint is being deleted with unrestored values. See the following logs for the specific values in question. To silence these warnings, use `status.expect_partial()`. See https://www.tensorflow.org/api_docs/python/tf/train/Checkpoint#restorefor details about the status object returned by the restore function.
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.1
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.2
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.3
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.4
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.5
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.6
Epoch 1/200
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.7
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.8
```

```
196/196 [==============================] - 1s 4ms/step - loss: 0.6037 - binary_accuracy: 0.6656 - val_loss: 0.5707 - val_binary_accuracy: 0.6968
Epoch 2/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5668 - binary_accuracy: 0.6987 - val_loss: 0.5623 - val_binary_accuracy: 0.7055
Epoch 3/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5624 - binary_accuracy: 0.7042 - val_loss: 0.5657 - val_binary_accuracy: 0.7024
Epoch 4/200
196/196 [==============================] - 1s 4ms/step - loss: 0.5622 - binary_accuracy: 0.7052 - val_loss: 0.5593 - val_binary_accuracy: 0.7058
Epoch 5/200
196/196 [==============================] - 1s 4ms/step - loss: 0.5596 - binary_accuracy: 0.7066 - val_loss: 0.5572 - val_binary_accuracy: 0.7088
Epoch 6/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5598 - binary_accuracy: 0.7063 - val_loss: 0.5568 - val_binary_accuracy: 0.7096
Epoch 7/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5568 - binary_accuracy: 0.7096 - val_loss: 0.5569 - val_binary_accuracy: 0.7071
Epoch 8/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5558 - binary_accuracy: 0.7099 - val_loss: 0.5546 - val_binary_accuracy: 0.7095
Epoch 9/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5562 - binary_accuracy: 0.7102 - val_loss: 0.5543 - val_binary_accuracy: 0.7110
Epoch 10/200
196/196 [==============================] - 1s 4ms/step - loss: 0.5538 - binary_accuracy: 0.7109 - val_loss: 0.5531 - val_binary_accuracy: 0.7110
Epoch 11/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5529 - binary_accuracy: 0.7127 - val_loss: 0.5530 - val_binary_accuracy: 0.7095
Epoch 12/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5527 - binary_accuracy: 0.7109 - val_loss: 0.5537 - val_binary_accuracy: 0.7110
Epoch 13/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5507 - binary_accuracy: 0.7141 - val_loss: 0.5585 - val_binary_accuracy: 0.7086
Epoch 14/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5504 - binary_accuracy: 0.7135 - val_loss: 0.5490 - val_binary_accuracy: 0.7158
Epoch 15/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5491 - binary_accuracy: 0.7149 - val_loss: 0.5499 - val_binary_accuracy: 0.7119
Epoch 16/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5466 - binary_accuracy: 0.7164 - val_loss: 0.5510 - val_binary_accuracy: 0.7142
Epoch 17/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5470 - binary_accuracy: 0.7164 - val_loss: 0.5479 - val_binary_accuracy: 0.7160
Epoch 18/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5456 - binary_accuracy: 0.7177 - val_loss: 0.5458 - val_binary_accuracy: 0.7183
Epoch 19/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5450 - binary_accuracy: 0.7178 - val_loss: 0.5508 - val_binary_accuracy: 0.7133
Epoch 20/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5448 - binary_accuracy: 0.7187 - val_loss: 0.5453 - val_binary_accuracy: 0.7195
Epoch 21/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5437 - binary_accuracy: 0.7188 - val_loss: 0.5460 - val_binary_accuracy: 0.7193
Epoch 22/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5430 - binary_accuracy: 0.7192 - val_loss: 0.5436 - val_binary_accuracy: 0.7190
Epoch 23/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5431 - binary_accuracy: 0.7192 - val_loss: 0.5456 - val_binary_accuracy: 0.7195
Epoch 24/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5433 - binary_accuracy: 0.7199 - val_loss: 0.5443 - val_binary_accuracy: 0.7202
Epoch 25/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5420 - binary_accuracy: 0.7204 - val_loss: 0.5438 - val_binary_accuracy: 0.7209
Epoch 26/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5416 - binary_accuracy: 0.7198 - val_loss: 0.5432 - val_binary_accuracy: 0.7221
Epoch 27/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5409 - binary_accuracy: 0.7214 - val_loss: 0.5424 - val_binary_accuracy: 0.7220
Epoch 28/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5411 - binary_accuracy: 0.7206 - val_loss: 0.5462 - val_binary_accuracy: 0.7170
Epoch 29/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5416 - binary_accuracy: 0.7192 - val_loss: 0.5439 - val_binary_accuracy: 0.7195
Epoch 30/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5395 - binary_accuracy: 0.7216 - val_loss: 0.5422 - val_binary_accuracy: 0.7221
Epoch 31/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5394 - binary_accuracy: 0.7219 - val_loss: 0.5411 - val_binary_accuracy: 0.7236
Epoch 32/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5390 - binary_accuracy: 0.7221 - val_loss: 0.5413 - val_binary_accuracy: 0.7219
Epoch 33/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5383 - binary_accuracy: 0.7233 - val_loss: 0.5401 - val_binary_accuracy: 0.7240
Epoch 34/200
196/196 [==============================] - 1s 4ms/step - loss: 0.5379 - binary_accuracy: 0.7226 - val_loss: 0.5418 - val_binary_accuracy: 0.7220
Epoch 35/200
196/196 [==============================] - 1s 4ms/step - loss: 0.5389 - binary_accuracy: 0.7218 - val_loss: 0.5402 - val_binary_accuracy: 0.7257
Epoch 36/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5377 - binary_accuracy: 0.7231 - val_loss: 0.5414 - val_binary_accuracy: 0.7215
Epoch 37/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5372 - binary_accuracy: 0.7235 - val_loss: 0.5413 - val_binary_accuracy: 0.7239
Epoch 38/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5375 - binary_accuracy: 0.7226 - val_loss: 0.5393 - val_binary_accuracy: 0.7237
786/786 [==============================] - 1s 1ms/step
ROC AUC Score: 0.7822921094652046
```



Get NN_campaign result

In [ ]:
```
# We will use
S_train_whole1 = cleaned_data.drop(columns=['purchase','Promotion'])
y_pred = best_model.predict(S_train_whole1)

# Flatten y_pred to make it 1-dimensional
y_pred = y_pred.flatten()
y_pred
```

```
7827/7827 [==============================] - 10s 1ms/step
```

Out[ ]:
```
array([0.39779598, 0.21099567, 0.10400918, ..., 0.6966119 , 0.77933466,
       0.18565813], dtype=float32)
```

In [ ]:
```
S_train_whole_final = cleaned_data.copy()
S_train_whole_final['NN_camp'] = y_pred
S_train_whole_final
```

Out[ ]:

	Promotion	V2	V3	purchase	V1_0	V1_1	V1_2	V1_3	V4_1	V4_2	...	V5_2	V5_3	V5_4	V6_1	V6_2	V6_3	V6_4	V7_1	V7_2	NN_camp
0	0	30.443518	-1.165083	0	0	0	1	0	1	0	...	0	0	0	0	0	1	0	0	1	0.397796
1	0	32.159350	-0.645617	0	0	0	0	1	0	1	...	0	1	0	0	1	0	0	0	1	0.210996
2	0	30.431659	0.133583	0	0	0	1	0	1	0	...	0	0	0	0	0	0	1	0	1	0.104009
3	0	26.588914	-0.212728	0	1	0	0	0	0	1	...	0	0	0	0	0	0	1	0	1	0.415342
4	1	28.044331	-0.385883	0	0	0	0	1	1	0	...	0	0	0	0	1	0	0	0	1	0.071641
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
250443	1	22.492320	0.795937	0	0	0	1	0	0	1	...	1	0	0	1	0	0	0	1	0	0.517117
250444	1	29.660005	1.264919	0	0	1	0	0	0	1	...	1	0	0	1	0	0	0	1	0	0.670246
250445	1	28.862809	0.963575	0	1	0	0	0	0	1	...	0	1	0	0	1	0	0	1	0	0.696612
250446	1	35.587445	0.592060	0	0	1	0	0	1	0	...	1	0	0	0	0	1	0	1	0	0.779335
250447	1	35.107513	1.511598	0	0	0	1	0	0	1	...	0	0	0	0	0	1	0	0	1	0.185658

250448 rows × 21 columns

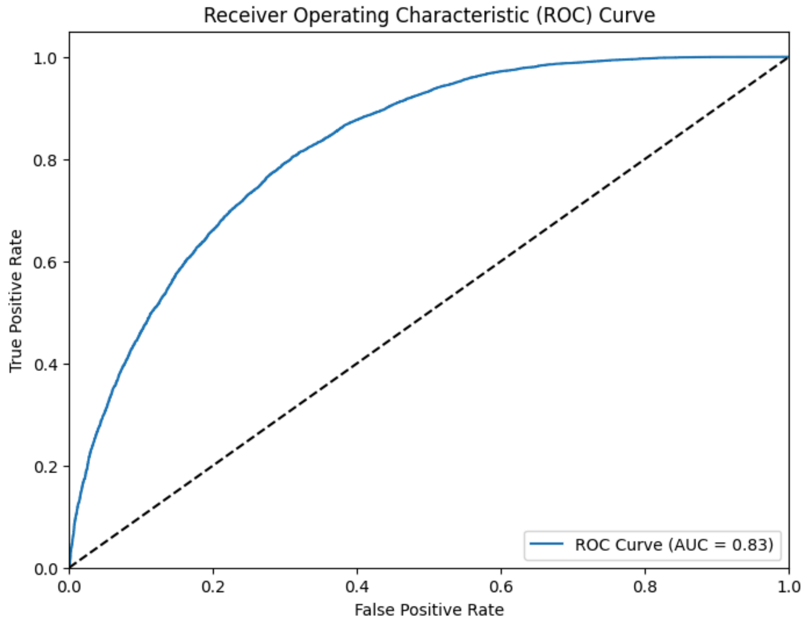
NoCampaign NN model

In [ ]:
```
# NN with no_Campaign

features_num = ['V2', 'V3']
features_cat = ['V1_0', 'V1_1', 'V1_2', 'V1_3',
       'V4_1', 'V4_2', 'V5_1', 'V5_2', 'V5_3', 'V5_4', 'V6_1', 'V6_2', 'V6_3',
       'V6_4', 'V7_1', 'V7_2']


X_train1 = S_train_nocam_x.drop(columns=['Promotion'])
X_valid1 = S_test_nocam_x.drop(columns=['Promotion'])
y_train1 = S_train_nocam_y
y_valid1 = S_test_nocam_y

input_shape1 = [X_train1.shape[1]]
print("Input shape: {}".format(input_shape1))


# Building the deep learning model
model1 = keras.Sequential([
    layers.Dense(units=64, activation='relu', input_shape= input_shape1),
    layers.Dense(units=64, activation='relu'),
    layers.Dense(units=1, activation='sigmoid')
```

```
])

# Compiling the model
model1.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['binary_accuracy']
)

# This will run the model and plot the learning curve
early_stopping1 = keras.callbacks.EarlyStopping(
    patience=5,
    min_delta=0.001,
    restore_best_weights=True,
)

history1 = model1.fit(
    X_train1, y_train1,
    validation_data=(X_valid1, y_valid1),
    batch_size=512,
    epochs=200,
    callbacks=[early_stopping1],
)

y_pred1 = model1.predict(X_valid1)

# Calculate the ROC curve
fpr1, tpr1, thresholds1 = roc_curve(y_valid1, y_pred1)

# Calculate the ROC AUC score
roc_auc1 = roc_auc_score(y_valid1, y_pred1)

print("ROC AUC Score:", roc_auc1)

# Plot the ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr1, tpr1, label=f'ROC Curve (AUC = {roc_auc1:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

```
Input shape: [18]
Epoch 1/200
196/196 [==============================] - 1s 4ms/step - loss: 0.5957 - binary_accuracy: 0.6778 - val_loss: 0.5531 - val_binary_accuracy: 0.7170
Epoch 2/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5464 - binary_accuracy: 0.7164 - val_loss: 0.5432 - val_binary_accuracy: 0.7195
Epoch 3/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5414 - binary_accuracy: 0.7204 - val_loss: 0.5402 - val_binary_accuracy: 0.7225
Epoch 4/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5375 - binary_accuracy: 0.7226 - val_loss: 0.5387 - val_binary_accuracy: 0.7239
Epoch 5/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5355 - binary_accuracy: 0.7231 - val_loss: 0.5374 - val_binary_accuracy: 0.7214
Epoch 6/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5355 - binary_accuracy: 0.7244 - val_loss: 0.5379 - val_binary_accuracy: 0.7244
Epoch 7/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5351 - binary_accuracy: 0.7246 - val_loss: 0.5418 - val_binary_accuracy: 0.7235
Epoch 8/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5326 - binary_accuracy: 0.7263 - val_loss: 0.5339 - val_binary_accuracy: 0.7266
Epoch 9/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5330 - binary_accuracy: 0.7259 - val_loss: 0.5329 - val_binary_accuracy: 0.7260
Epoch 10/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5313 - binary_accuracy: 0.7274 - val_loss: 0.5347 - val_binary_accuracy: 0.7255
Epoch 11/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5322 - binary_accuracy: 0.7254 - val_loss: 0.5370 - val_binary_accuracy: 0.7239
Epoch 12/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5314 - binary_accuracy: 0.7257 - val_loss: 0.5331 - val_binary_accuracy: 0.7260
Epoch 13/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5304 - binary_accuracy: 0.7264 - val_loss: 0.5320 - val_binary_accuracy: 0.7258
Epoch 14/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5296 - binary_accuracy: 0.7283 - val_loss: 0.5423 - val_binary_accuracy: 0.7210
Epoch 15/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5288 - binary_accuracy: 0.7287 - val_loss: 0.5318 - val_binary_accuracy: 0.7293
Epoch 16/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5275 - binary_accuracy: 0.7276 - val_loss: 0.5309 - val_binary_accuracy: 0.7269
Epoch 17/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5275 - binary_accuracy: 0.7293 - val_loss: 0.5380 - val_binary_accuracy: 0.7217
Epoch 18/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5269 - binary_accuracy: 0.7287 - val_loss: 0.5293 - val_binary_accuracy: 0.7290
Epoch 19/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5258 - binary_accuracy: 0.7298 - val_loss: 0.5276 - val_binary_accuracy: 0.7293
Epoch 20/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5250 - binary_accuracy: 0.7307 - val_loss: 0.5289 - val_binary_accuracy: 0.7291
Epoch 21/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5237 - binary_accuracy: 0.7317 - val_loss: 0.5255 - val_binary_accuracy: 0.7311
Epoch 22/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5243 - binary_accuracy: 0.7312 - val_loss: 0.5254 - val_binary_accuracy: 0.7300
Epoch 23/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5233 - binary_accuracy: 0.7325 - val_loss: 0.5252 - val_binary_accuracy: 0.7324
Epoch 24/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5223 - binary_accuracy: 0.7333 - val_loss: 0.5242 - val_binary_accuracy: 0.7318
Epoch 25/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5218 - binary_accuracy: 0.7336 - val_loss: 0.5231 - val_binary_accuracy: 0.7323
Epoch 26/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5216 - binary_accuracy: 0.7334 - val_loss: 0.5257 - val_binary_accuracy: 0.7309
Epoch 27/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5205 - binary_accuracy: 0.7343 - val_loss: 0.5226 - val_binary_accuracy: 0.7322
Epoch 28/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5194 - binary_accuracy: 0.7343 - val_loss: 0.5228 - val_binary_accuracy: 0.7357
Epoch 29/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5187 - binary_accuracy: 0.7352 - val_loss: 0.5207 - val_binary_accuracy: 0.7363
Epoch 30/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5190 - binary_accuracy: 0.7346 - val_loss: 0.5254 - val_binary_accuracy: 0.7292
Epoch 31/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5173 - binary_accuracy: 0.7361 - val_loss: 0.5201 - val_binary_accuracy: 0.7328
Epoch 32/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5170 - binary_accuracy: 0.7362 - val_loss: 0.5193 - val_binary_accuracy: 0.7331
Epoch 33/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5159 - binary_accuracy: 0.7367 - val_loss: 0.5191 - val_binary_accuracy: 0.7348
Epoch 34/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5156 - binary_accuracy: 0.7370 - val_loss: 0.5175 - val_binary_accuracy: 0.7351
Epoch 35/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5152 - binary_accuracy: 0.7373 - val_loss: 0.5173 - val_binary_accuracy: 0.7352
Epoch 36/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5148 - binary_accuracy: 0.7380 - val_loss: 0.5167 - val_binary_accuracy: 0.7357
Epoch 37/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5128 - binary_accuracy: 0.7394 - val_loss: 0.5141 - val_binary_accuracy: 0.7411
Epoch 38/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5131 - binary_accuracy: 0.7379 - val_loss: 0.5138 - val_binary_accuracy: 0.7393
Epoch 39/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5120 - binary_accuracy: 0.7396 - val_loss: 0.5122 - val_binary_accuracy: 0.7413
Epoch 40/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5119 - binary_accuracy: 0.7406 - val_loss: 0.5133 - val_binary_accuracy: 0.7395
Epoch 41/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5101 - binary_accuracy: 0.7405 - val_loss: 0.5126 - val_binary_accuracy: 0.7406
Epoch 42/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5101 - binary_accuracy: 0.7407 - val_loss: 0.5111 - val_binary_accuracy: 0.7441
Epoch 43/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5093 - binary_accuracy: 0.7413 - val_loss: 0.5123 - val_binary_accuracy: 0.7403
Epoch 44/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5083 - binary_accuracy: 0.7414 - val_loss: 0.5145 - val_binary_accuracy: 0.7403
Epoch 45/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5079 - binary_accuracy: 0.7419 - val_loss: 0.5087 - val_binary_accuracy: 0.7423
Epoch 46/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5070 - binary_accuracy: 0.7427 - val_loss: 0.5075 - val_binary_accuracy: 0.7426
Epoch 47/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5060 - binary_accuracy: 0.7427 - val_loss: 0.5078 - val_binary_accuracy: 0.7409
Epoch 48/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5061 - binary_accuracy: 0.7423 - val_loss: 0.5063 - val_binary_accuracy: 0.7449
Epoch 49/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5049 - binary_accuracy: 0.7440 - val_loss: 0.5087 - val_binary_accuracy: 0.7422
Epoch 50/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5047 - binary_accuracy: 0.7440 - val_loss: 0.5057 - val_binary_accuracy: 0.7449
Epoch 51/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5036 - binary_accuracy: 0.7442 - val_loss: 0.5040 - val_binary_accuracy: 0.7459
Epoch 52/200
```

```
196/196 [==============================] - 1s 3ms/step - loss: 0.5042 - binary_accuracy: 0.7436 - val_loss: 0.5040 - val_binary_accuracy: 0.7459
Epoch 53/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5020 - binary_accuracy: 0.7459 - val_loss: 0.5041 - val_binary_accuracy: 0.7441
Epoch 54/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5019 - binary_accuracy: 0.7466 - val_loss: 0.5061 - val_binary_accuracy: 0.7453
Epoch 55/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5007 - binary_accuracy: 0.7469 - val_loss: 0.5053 - val_binary_accuracy: 0.7449
Epoch 56/200
196/196 [==============================] - 1s 3ms/step - loss: 0.4997 - binary_accuracy: 0.7476 - val_loss: 0.5038 - val_binary_accuracy: 0.7466
781/781 [==============================] - 1s 1ms/step
ROC AUC Score: 0.8257237148745478
```



NoCampaign NN model - Tuning

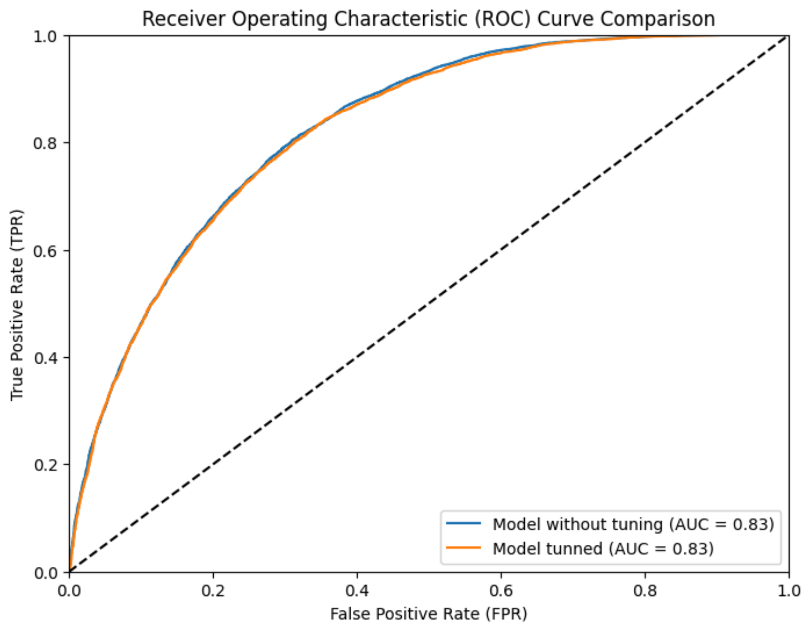
In [ ]:
```python
# Tuning - NN with noCampaign
input_shape = (X_train1.shape[1])  # Creating a tuple with a single element

# Define a function that builds your Keras model with hyperparameters
def build_model(hp):
    model = keras.Sequential()
    model.add(layers.Dense(units=hp.Int('units', min_value=32, max_value=512, step=32), activation='relu', input_shape=(input_shape,)))
    model.add(layers.Dense(1, activation='sigmoid'))  # Output layer for binary classification
    model.compile(
        optimizer=hp.Choice('optimizer', values=['adam', 'rmsprop', 'sgd']),
        loss='binary_crossentropy',
        metrics=['binary_accuracy']
    )
    return model

tuner1 = RandomSearch(
    build_model,
    objective='binary_accuracy',
    max_trials=10,  # Number of different hyperparameter combinations to try
    directory='my_dir_nocam',  # Directory where logs and results will be stored
    project_name='my_project_nocam'  # Name for the tuning project
)

tuner1.search(X_train1, y_train1, epochs=10, validation_data=(X_valid1, y_valid1))

#best_model1 = tuner1.get_best_models(num_models=1)[0]
best_hyperparameters1 = tuner1.get_best_hyperparameters(num_trials=1)[0]
#best_model1.summary()
```

```
Reloading Tuner from my_dir_nocam/my_project_nocam/tuner0.json
```

In [ ]:
```python
# Build the best model using the best hyperparameters
best_model1 = build_model(best_hyperparameters1)

# Train the best model with your training data
history_tuned1 = best_model1.fit(
    X_train1, y_train1,
    validation_data=(X_valid1, y_valid1),
    batch_size=512,
    epochs=200,
    callbacks=[early_stopping1],
)

y_pred_tuned1 = best_model1.predict(X_valid1)

history_df_tuned1 = pd.DataFrame(history_tuned1.history)

# Calculate the ROC curve
fpr11, tpr11, thresholds11 = roc_curve(y_valid1, y_pred_tuned1)

# Calculate the ROC AUC score
roc_auc_tuned11 = roc_auc_score(y_valid1, y_pred1)

print("ROC AUC Score:", roc_auc_tuned11)
```

```python
# Print both models to see if it has really improved
# Create a plot to compare the ROC curves
plt.figure(figsize=(8, 6))
plt.plot(fpr1, tpr1, label=f'Model without tuning (AUC = {roc_auc1:.2f})')
plt.plot(fpr11, tpr11, label=f'Model tunned (AUC = {roc_auc_tuned11:.2f})')
plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line for reference
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Receiver Operating Characteristic (ROC) Curve Comparison')
plt.legend(loc='lower right')
plt.show()
```

```
Epoch 1/200
196/196 [==============================] - 1s 4ms/step - loss: 0.5869 - binary_accuracy: 0.6861 - val_loss: 0.5493 - val_binary_accuracy: 0.7176
Epoch 2/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5437 - binary_accuracy: 0.7198 - val_loss: 0.5407 - val_binary_accuracy: 0.7228
Epoch 3/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5376 - binary_accuracy: 0.7236 - val_loss: 0.5379 - val_binary_accuracy: 0.7205
Epoch 4/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5352 - binary_accuracy: 0.7246 - val_loss: 0.5353 - val_binary_accuracy: 0.7255
Epoch 5/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5331 - binary_accuracy: 0.7257 - val_loss: 0.5337 - val_binary_accuracy: 0.7254
Epoch 6/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5332 - binary_accuracy: 0.7258 - val_loss: 0.5322 - val_binary_accuracy: 0.7259
Epoch 7/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5306 - binary_accuracy: 0.7281 - val_loss: 0.5310 - val_binary_accuracy: 0.7260
Epoch 8/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5285 - binary_accuracy: 0.7287 - val_loss: 0.5304 - val_binary_accuracy: 0.7296
Epoch 9/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5286 - binary_accuracy: 0.7284 - val_loss: 0.5291 - val_binary_accuracy: 0.7284
Epoch 10/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5270 - binary_accuracy: 0.7301 - val_loss: 0.5278 - val_binary_accuracy: 0.7302
Epoch 11/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5256 - binary_accuracy: 0.7308 - val_loss: 0.5373 - val_binary_accuracy: 0.7205
Epoch 12/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5250 - binary_accuracy: 0.7322 - val_loss: 0.5260 - val_binary_accuracy: 0.7304
Epoch 13/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5239 - binary_accuracy: 0.7318 - val_loss: 0.5269 - val_binary_accuracy: 0.7301
Epoch 14/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5228 - binary_accuracy: 0.7337 - val_loss: 0.5257 - val_binary_accuracy: 0.7314
Epoch 15/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5223 - binary_accuracy: 0.7337 - val_loss: 0.5263 - val_binary_accuracy: 0.7290
Epoch 16/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5226 - binary_accuracy: 0.7324 - val_loss: 0.5231 - val_binary_accuracy: 0.7339
Epoch 17/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5209 - binary_accuracy: 0.7335 - val_loss: 0.5229 - val_binary_accuracy: 0.7338
Epoch 18/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5196 - binary_accuracy: 0.7345 - val_loss: 0.5238 - val_binary_accuracy: 0.7329
Epoch 19/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5185 - binary_accuracy: 0.7358 - val_loss: 0.5276 - val_binary_accuracy: 0.7276
Epoch 20/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5179 - binary_accuracy: 0.7361 - val_loss: 0.5259 - val_binary_accuracy: 0.7291
Epoch 21/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5165 - binary_accuracy: 0.7365 - val_loss: 0.5179 - val_binary_accuracy: 0.7397
Epoch 22/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5155 - binary_accuracy: 0.7391 - val_loss: 0.5188 - val_binary_accuracy: 0.7373
Epoch 23/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5152 - binary_accuracy: 0.7386 - val_loss: 0.5204 - val_binary_accuracy: 0.7353
Epoch 24/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5149 - binary_accuracy: 0.7382 - val_loss: 0.5202 - val_binary_accuracy: 0.7327
Epoch 25/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5138 - binary_accuracy: 0.7398 - val_loss: 0.5183 - val_binary_accuracy: 0.7353
Epoch 26/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5136 - binary_accuracy: 0.7396 - val_loss: 0.5151 - val_binary_accuracy: 0.7418
Epoch 27/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5134 - binary_accuracy: 0.7390 - val_loss: 0.5180 - val_binary_accuracy: 0.7367
Epoch 28/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5115 - binary_accuracy: 0.7404 - val_loss: 0.5142 - val_binary_accuracy: 0.7422
Epoch 29/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5109 - binary_accuracy: 0.7409 - val_loss: 0.5141 - val_binary_accuracy: 0.7417
Epoch 30/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5104 - binary_accuracy: 0.7416 - val_loss: 0.5115 - val_binary_accuracy: 0.7415
Epoch 31/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5093 - binary_accuracy: 0.7421 - val_loss: 0.5160 - val_binary_accuracy: 0.7371
Epoch 32/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5087 - binary_accuracy: 0.7425 - val_loss: 0.5127 - val_binary_accuracy: 0.7425
Epoch 33/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5078 - binary_accuracy: 0.7432 - val_loss: 0.5114 - val_binary_accuracy: 0.7449
Epoch 34/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5068 - binary_accuracy: 0.7450 - val_loss: 0.5089 - val_binary_accuracy: 0.7425
Epoch 35/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5073 - binary_accuracy: 0.7438 - val_loss: 0.5102 - val_binary_accuracy: 0.7437
Epoch 36/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5071 - binary_accuracy: 0.7435 - val_loss: 0.5135 - val_binary_accuracy: 0.7373
Epoch 37/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5058 - binary_accuracy: 0.7444 - val_loss: 0.5256 - val_binary_accuracy: 0.7343
Epoch 38/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5059 - binary_accuracy: 0.7432 - val_loss: 0.5100 - val_binary_accuracy: 0.7401
Epoch 39/200
196/196 [==============================] - 1s 3ms/step - loss: 0.5042 - binary_accuracy: 0.7452 - val_loss: 0.5080 - val_binary_accuracy: 0.7399
781/781 [==============================] - 1s 1ms/step
ROC AUC Score: 0.8257237148745478
```



Get NN_nocampaign result

In [ ]:
```python
# There is no improvement, so we just use original model.
S_train_whole1 = cleaned_data.drop(columns=['purchase','Promotion']).copy()
y_pred1 = best_model1.predict(S_train_whole1)

# Flatten y_pred to make it 1-dimensional
y_pred1 = y_pred1.flatten()

S_train_whole_final['NN_nocamp'] = y_pred1
S_train_whole_final
```

```
7827/7827 [==============================] - 9s 1ms/step
```

Out[ ]:

	Promotion	V2	V3	purchase	V1_0	V1_1	V1_2	V1_3	V4_1	V4_2	...	V6_4	V7_1	V7_2	NN_camp	NN_nocamp	Uplift_Score	Uplift_Score_xgb	Uplift_
0	0	30.443518	-1.165083	0	0	0	1	0	1	0	...	0	0	1	0.397796	0.403136	-0.005340	0.128103	-
1	0	32.159350	-0.645617	0	0	0	0	1	0	1	...	0	0	1	0.210996	0.091249	0.119747	-0.046202	-
2	0	30.431659	0.133583	0	0	0	1	0	1	0	...	1	0	1	0.104009	0.296777	-0.192768	-0.074393	
3	0	26.588914	-0.212728	0	1	0	0	0	0	1	...	1	0	1	0.415342	0.085881	0.329461	-0.078622	
4	1	28.044331	-0.385883	0	0	0	0	1	1	0	...	0	0	1	0.071641	0.320202	-0.248561	0.109687	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
250443	1	22.492320	0.795937	0	0	0	1	0	0	1	...	0	1	0	0.517117	0.543745	-0.026628	-0.033232	
250444	1	29.660005	1.264919	0	0	1	0	0	0	1	...	0	1	0	0.670246	0.719154	-0.048908	0.223293	-
250445	1	28.862809	0.963575	0	1	0	0	0	0	1	...	0	1	0	0.696612	0.595317	0.101295	-0.004477	
250446	1	35.587445	0.592060	0	0	1	0	0	1	0	...	0	1	0	0.779335	0.894112	-0.114777	-0.053450	
250447	1	35.107513	1.511598	0	0	0	1	0	0	1	...	0	0	1	0.185658	0.203728	-0.018070	-0.109484	-

250448 rows × 27 columns

In [ ]:
```python
# Uplifting Score

S_train_whole_final['Uplift_Score'] = S_train_whole_final['NN_camp'] - S_train_whole_final['NN_nocamp']

# Assuming S_train_whole_final is already defined and has a column 'Uplift_Score'
#S_train_whole_final['persuasive'] = S_train_whole_final['Uplift_Score'].apply(lambda x: 1 if x > 0 else 0)
S_train_whole_final
```

Out[ ]:

	Promotion	V2	V3	purchase	V1_0	V1_1	V1_2	V1_3	V4_1	V4_2	...	V6_4	V7_1	V7_2	NN_camp	NN_nocamp	Uplift_Score	Uplift_Score_xgb	Uplift_
0	0	30.443518	-1.165083	0	0	0	1	0	1	0	...	0	0	1	0.397796	0.403136	-0.005340	0.128103	-
1	0	32.159350	-0.645617	0	0	0	0	1	0	1	...	0	0	1	0.210996	0.091249	0.119747	-0.046202	-
2	0	30.431659	0.133583	0	0	0	1	0	1	0	...	1	0	1	0.104009	0.296777	-0.192768	-0.074393	
3	0	26.588914	-0.212728	0	1	0	0	0	0	1	...	1	0	1	0.415342	0.085881	0.329461	-0.078622	
4	1	28.044331	-0.385883	0	0	0	0	1	1	0	...	0	0	1	0.071641	0.320202	-0.248561	0.109687	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
250443	1	22.492320	0.795937	0	0	0	1	0	0	1	...	0	1	0	0.517117	0.543745	-0.026628	-0.033232	
250444	1	29.660005	1.264919	0	0	1	0	0	0	1	...	0	1	0	0.670246	0.719154	-0.048908	0.223293	-
250445	1	28.862809	0.963575	0	1	0	0	0	0	1	...	0	1	0	0.696612	0.595317	0.101295	-0.004477	
250446	1	35.587445	0.592060	0	0	1	0	0	1	0	...	0	1	0	0.779335	0.894112	-0.114777	-0.053450	
250447	1	35.107513	1.511598	0	0	0	1	0	0	1	...	0	0	1	0.185658	0.203728	-0.018070	-0.109484	-

250448 rows × 27 columns

In [ ]:
```python
S_train_whole_final['Uplift_Score_log']= uplift_scores_log
S_train_whole_final['Uplift_Score_xgb']= uplift_scores_xgb
S_train_whole_final['Uplift_Score_rf']= uplift_scores_rf
```

In [ ]:
```python
S_train_whole_final
```

Out[ ]:

	Promotion	V2	V3	purchase	V1_0	V1_1	V1_2	V1_3	V4_1	V4_2	...	V7_1	V7_2	NN_camp	NN_nocamp	Uplift_Score	Uplift_Score_xgb	Uplift_Score_
0	0	30.443518	-1.165083	0	0	0	1	0	1	0	...	0	1	0.397796	0.403136	-0.005340	0.128103	-0.0597
1	0	32.159350	-0.645617	0	0	0	0	1	0	1	...	0	1	0.210996	0.091249	0.119747	-0.046202	-0.0780
2	0	30.431659	0.133583	0	0	0	1	0	1	0	...	0	1	0.104009	0.296777	-0.192768	-0.074393	0.0196
3	0	26.588914	-0.212728	0	1	0	0	0	0	1	...	0	1	0.415342	0.085881	0.329461	-0.078622	-0.1595
4	1	28.044331	-0.385883	0	0	0	0	1	1	0	...	0	1	0.071641	0.320202	-0.248561	0.109687	0.1260
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
250443	1	22.492320	0.795937	0	0	0	1	0	0	1	...	1	0	0.517117	0.543745	-0.026628	-0.033232	0.0537
250444	1	29.660005	1.264919	0	0	1	0	0	0	1	...	1	0	0.670246	0.719154	-0.048908	0.223293	-0.0529
250445	1	28.862809	0.963575	0	1	0	0	0	0	1	...	1	0	0.696612	0.595317	0.101295	-0.004477	0.0944
250446	1	35.587445	0.592060	0	0	1	0	0	1	0	...	1	0	0.779335	0.894112	-0.114777	-0.053450	-0.2517
250447	1	35.107513	1.511598	0	0	0	1	0	0	1	...	0	1	0.185658	0.203728	-0.018070	-0.109484	-0.0895

250448 rows × 28 columns

In [ ]:
```python
from sklift.metrics import qini_auc_score
# Calculating the Qini coefficient using scikit-uplift
qini_score_log = qini_auc_score(S_train_whole_final['purchase'], S_train_whole_final['Uplift_Score_log'], S_train_whole_final['purchase'])
qini_score_xgb = qini_auc_score(S_train_whole_final['purchase'], S_train_whole_final['Uplift_Score_xgb'], S_train_whole_final['purchase'])
qini_score_rf = qini_auc_score(S_train_whole_final['purchase'], S_train_whole_final['Uplift_Score_rf'], S_train_whole_final['purchase'])
qini_score_nn = qini_auc_score(S_train_whole_final['purchase'], S_train_whole_final['Uplift_Score'], S_train_whole_final['purchase'])


print("Qini AUC Score for Logistic Regression:", qini_score_log)
print("Qini AUC Score for XGBoost:", qini_score_xgb)
print("Qini AUC Score for Random Forest:", qini_score_rf)
print("Qini AUC Score for Neural Network:", qini_score_nn)
```

```
Qini AUC Score for Logistic Regression: 0.05323654823053096
Qini AUC Score for XGBoost: 0.06702339557643668
Qini AUC Score for Random Forest: -0.028638173015179565
Qini AUC Score for Neural Network: -0.048151272046538726
```

In [ ]:
```python
def calculate_IRR(df, response_column, score_column):
    # Define thresholds or decide on a method to segment the customers based on uplift score
    # For simplicity, let's consider the top decile as the treatment group
    threshold = df[score_column].quantile(0.9)
    treated = df[df[score_column] >= threshold]
    control = df[df[score_column] < threshold]

    # Calculate response rates
    response_rate_treated = treated[response_column].mean()
    response_rate_control = control[response_column].mean()

    # Calculate IRR
    irr = response_rate_treated - response_rate_control
    return irr

irr_log = calculate_IRR(S_train_whole_final, 'purchase', 'Uplift_Score_log')
irr_xgb = calculate_IRR(S_train_whole_final, 'purchase', 'Uplift_Score_xgb')
irr_rf = calculate_IRR(S_train_whole_final, 'purchase', 'Uplift_Score_rf')
irr_nn = calculate_IRR(S_train_whole_final, 'purchase', 'Uplift_Score')


print(f"IRR for Logistic Regression: {irr_log:.4f}")
print(f"IRR for XGBoost: {irr_xgb:.4f}")
print(f"IRR for Random Forest: {irr_rf:.4f}")
print(f"IRR for NN: {irr_nn:.4f}")
```

```
IRR for Logistic Regression: 0.1108
IRR for XGBoost: -0.0196
IRR for Random Forest: 0.4588
IRR for NN: -0.0049
```

In [ ]:
```
import numpy as np
from scipy.optimize import minimize
from sklift.metrics import qini_auc_score

# Assuming you have functions to calculate IRR and uplift scores already defined
def calculate_combined_IRR(weights):
    final_uplift_score = weights[0] * S_train_whole_final['Uplift_Score_log'] + \
                         weights[1] * S_train_whole_final['Uplift_Score_xgb'] + \
                         weights[2] * S_train_whole_final['Uplift_Score_rf'] + \
                         weights[3] * S_train_whole_final['Uplift_Score']
    S_train_whole_final['Final_Uplift_Score'] = final_uplift_score
    irr = calculate_IRR(S_train_whole_final, 'purchase', 'Final_Uplift_Score')
    qini = qini_auc_score(S_train_whole_final['purchase'], S_train_whole_final['Final_Uplift_Score'], S_train_whole_final['purchase'])
    # Define how you want to combine IRR and Qini score, e.g., average them
    return -(irr + qini)  # negative sign because we want to maximize

# Constraint: sum of weights = 1
cons = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1})

# Bounds for each weight to be between 0 and 1
bounds = [(0, 1)] * 4

# Initial guess for weights
initial_weights = [0.25, 0.25, 0.25, 0.25]

# Perform the minimization
result = minimize(calculate_combined_IRR, initial_weights, method='SLSQP', bounds=bounds, constraints=cons)

# The optimal weights
print("Optimal weights:", result.x)
```

```
Optimal weights: [0.25001355 0.24999748 0.24999235 0.24999662]
```

In [ ]:
```
# Let's say these are your optimal weights obtained from the optimization procedure
optimal_weights = [0.2500042, 0.2499993, 0.24999774, 0.24999903]

# Calculate the final uplift score using the optimized weights
S_train_whole_final['Final_Uplift_Score'] = (
    optimal_weights[0] * S_train_whole_final['Uplift_Score_log'] +
    optimal_weights[1] * S_train_whole_final['Uplift_Score_xgb'] +
    optimal_weights[2] * S_train_whole_final['Uplift_Score_rf'] +
    optimal_weights[3] * S_train_whole_final['Uplift_Score']  # Assuming this is the neural network uplift score
)

final_irr = calculate_IRR(S_train_whole_final, 'purchase', 'Final_Uplift_Score')

# Calculate the final Qini index
final_qini = qini_auc_score(
    S_train_whole_final['purchase'],
    S_train_whole_final['Final_Uplift_Score'],
    S_train_whole_final['Promotion'])

# Print out the final IRR and Qini index
print(f"Final IRR: {final_irr}")
print(f"Final Qini Index: {final_qini}")
```

```
Final IRR: 0.4250796788740919
Final Qini Index: 0.584268131050114
```

Meta-Learners

Base Learner

Listed above

T-Learner

Stage 1

Estimate the average outcomes $\mu_0(x)$ and $\mu_1(x)$:

$$\mu_0(x) = \mathbb{E}[Y(0) \mid X = x]$$
$$\mu_1(x) = \mathbb{E}[Y(1) \mid X = x]$$

using machine learning models.

Stage 2

Define the CATE estimate as:

$$\hat{\tau}(x) = \hat{\mu}_1(x) - \hat{\mu}_0(x)$$

In [ ]:
```
import pandas as pd
from sklearn.base import clone, BaseEstimator

class TLearner(BaseEstimator):
    def __init__(self, model):
        """
        Initialize the T-Learner with a given model.

        Parameters:
        - model: A machine learning model instance from sklearn.
        """
        self.model = model
```

```python
        self.model_0 = clone(model)
        self.model_1 = clone(model)
        self.is_fitted = False

    def fit(self, data, y, D, X):
        """
        Train the T-Learner models using the provided data.

        Parameters:
        - data: DataFrame containing the training data.
        - y: Name of the customer response column.
        - D: Name of the treatment indicator column.
        - X: List of feature column names.
        """
        # Split the data into treatment and control groups
        control_data = data[data[D] == 0]
        treatment_data = data[data[D] == 1]

        # Train the model on the control group
        self.model_0.fit(control_data[X], control_data[y])

        # Train the model on the treatment group
        self.model_1.fit(treatment_data[X], treatment_data[y])

        self.is_fitted = True
        return self

    def predict(self, data, X):
        """
        Predict the treatment effects using the trained models on new data.

        Parameters:
        - data: DataFrame containing the test data.
        - X: List of feature column names.

        Returns:
        - A DataFrame with the predicted treatment effects.
        """
        if not self.is_fitted:
            raise ValueError("This TLearner instance is not fitted yet. Call 'fit' with appropriate data.")

        # Predict outcomes using the control and treatment models
        data['mu_0_hat'] = self.model_0.predict(data[X])
        data['mu_1_hat'] = self.model_1.predict(data[X])

        # Calculate the treatment effect
        data['estimated_treatment_effect'] = data['mu_1_hat'] - data['mu_0_hat']

        return data[['mu_0_hat', 'mu_1_hat', 'estimated_treatment_effect']]

# Example usage:
# Assume 'model' is an instance of an sklearn regressor, such as sklearn.linear_model.LinearRegression()
# 'data' is a DataFrame with the necessary columns, and 'test_data' is your test DataFrame

#from sklearn.linear_model import LinearRegression
#t_learner = TLearner(LinearRegression())
#t_learner.fit(data=data, y='outcome', D='treatment', X=['feature1', 'feature2', 'feature3'])
#predictions = t_learner.predict(test_data, X=['feature1', 'feature2', 'feature3'])
```

X-Learner

Step 1:

Estimate $\mu_0(s)$ and $\mu_1(s)$ separately with any regression algorithms or supervised machine learning methods (same as T-learner);

Step 2:

Obtain the imputed treatment effects for individuals

$$\tilde{\Delta}_i^1 := R_i^1 - \hat{\mu}_0\left(S_i^1\right), \quad \tilde{\Delta}_i^0 := \hat{\mu}_1\left(S_i^0\right) - R_i^0.$$

Step 3:

Fit the imputed treatment effects to obtain $\hat{\tau}_1(s) := \mathbb{E}\left[\tilde{\Delta}_i^1 \mid S = s\right]$ and $\hat{\tau}_0(s) := \mathbb{E}\left[\tilde{\Delta}_i^0 \mid S = s\right]$;

Step 4: The final HTE estimator is given by

$$\hat{\tau}_{\text{X-learner}}(s) = g(s)\hat{\tau}_0(s) + (1 - g(s))\hat{\tau}_1(s),$$

where $g(s)$ is a weight function between $[0, 1]$. A possible way is to use the propensity score model as an estimate of $g(s)$.

In [ ]:
```python
import numpy as np
from sklearn.base import BaseEstimator, clone
from sklearn.linear_model import LogisticRegressionCV

class XLearner(BaseEstimator):
    def __init__(self, model):
        """
        Initialize the X-Learner with a given base model.

        Parameters:
        - model: A machine learning model instance.
        """
        self.model_0 = clone(model)
        self.model_1 = clone(model)
        self.propensity_model = LogisticRegressionCV()
```

```python
        self.model_tau_0 = clone(model)
        self.model_tau_1 = clone(model)
        self.is_fitted = False

    def fit(self, data, y, D, X):
        """
        Train the X-Learner models using the provided data.

        Parameters:
        - data: DataFrame containing the training data.
        - y: String name of the outcome variable column.
        - D: String name of the treatment indicator column.
        - X: List of feature column names.
        """
        control_data = data[data[D] == 0]
        treatment_data = data[data[D] == 1]

        # Step 1: Estimate mu_0 and mu_1
        self.model_0.fit(control_data[X], control_data[y])
        self.model_1.fit(treatment_data[X], treatment_data[y])

        # Predict outcomes using the control and treatment models
        control_data['mu_0_hat'] = self.model_0.predict(control_data[X])
        treatment_data['mu_1_hat'] = self.model_1.predict(treatment_data[X])

        # Step 2: Obtain the imputed treatment effects
        control_data['imputed_treatment_effect'] = control_data[y] - control_data['mu_0_hat']
        treatment_data['imputed_treatment_effect'] = treatment_data[y] - treatment_data['mu_1_hat']

        # Step 3: Fit the imputed treatment effects to estimate tau_0 and tau_1
        self.model_tau_0.fit(control_data[X], control_data['imputed_treatment_effect'])
        self.model_tau_1.fit(treatment_data[X], treatment_data['imputed_treatment_effect'])

        # Step 4 (a part of it): Fit a propensity model
        self.propensity_model.fit(data[X], data[D])

        self.is_fitted = True
        return self

    def predict(self, data, X):
        """
        Predict the treatment effects using the trained models on new data.

        Parameters:
        - data: DataFrame containing the test data.
        - X: List of feature column names.

        Returns:
        - A DataFrame with the predicted treatment effects.
        """
        if not self.is_fitted:
            raise ValueError("This XLearner instance is not fitted yet. Call 'fit' with appropriate data.")

        # Step 4: Use the fitted propensity model to estimate the weight g(s)
        g = self.propensity_model.predict_proba(data[X])[:, 1]

        # Predict the treatment effects using the imputed models
        tau_0_hat = self.model_tau_0.predict(data[X])
        tau_1_hat = self.model_tau_1.predict(data[X])

        # Calculate the final heterogeneous treatment effect estimate
        data['estimated_treatment_effect'] = g * tau_0_hat + (1 - g) * tau_1_hat

        return data[['estimated_treatment_effect']]

# Example usage:
# from sklearn.ensemble import RandomForestRegressor
# x_learner = XLearner(RandomForestRegressor())
# x_learner.fit(data=df, y='outcome', D='treatment', X=['feature1', 'feature2'])
# predictions = x_learner.predict(test_data, X=['feature1', 'feature2'])
```